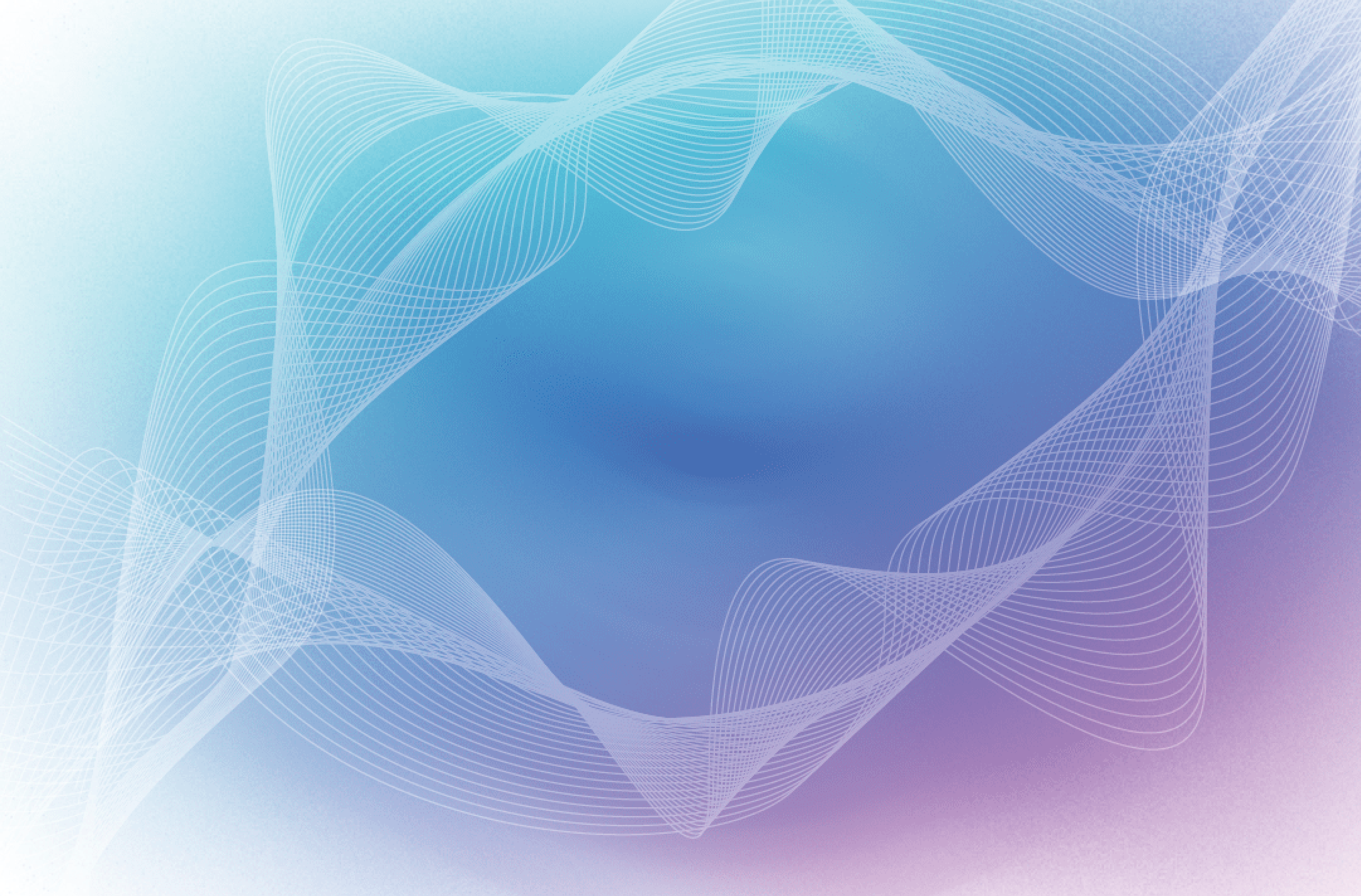


高等学校计算机教材建设立项项目

Python 程序设计与实践

——用计算思维解决问题

李莹　主编　　焦福菊　孙青　编著



清华大学出版社

高等学校计算机教材建设立项项目

Python 程序设计与实践

——用计算思维解决问题

李　莹　主　编
焦福菊　孙　青　编　著

清华大学出版社
北　京

内 容 简 介

本教材主要讲授 Python 程序设计知识，采用案例教学和问题驱动的撰写方法，注重实践思维、计算思维和创新思维等教育理念与教材内容的结合。本教材将知识点和实际应用相结合，以教学案例引出理论讲解。案例源于现实生活，旨在让读者理解实际问题被抽象化、模型化和程序化的全过程。内容涵盖 Python 应用的各个方面，以对比方式阐述人和计算机在解决问题时的异同，让读者理解计算思维的本质。教材在设计上由易到难，分别阐述计算机如何描述和处理现实世界中的各类事物，如何表示各类事物之间的关系，如何组织和优化程序结构等，使读者能够将程序设计和现实问题相关联。在讲解某一知识点时，横向延伸与之相关的各类知识点；在讲解某一个案例时，纵向扩展该案例所能实现的各种功能模块，使读者能够比较全面、深入地理解问题和掌握知识。教材穿插了一些技巧性、实用性的说明，并且对重要代码添加了注释。本教材免费提供与内容相配套的教学课件和各个案例的程序源代码。

本教材的内容涵盖范围较广，案例贴近实际，既可作为以 Python 为基础的程序设计类课程的配套教材，又可作为学习 Python 的很好的自学参考书，也适合各层次 Python 开发人员阅读参考。

图书在版编目(CIP)数据

Python 程序设计与实践：用计算思维解决问题/李莹主编；焦福菊，孙青编著. —北京：清华大学出版社，2018
ISBN 978-7-302-47389-3

Ⅰ. ①P…　Ⅱ. ①李… ②焦… ③孙…　Ⅲ. ①软件工具－程序设计　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2017)第 101920 号

责任编辑：张瑞庆
封面设计：何凤霞
责任校对：焦丽丽
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京鑫海金澳胶印有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张**：11　　**字　　数**：175 千字
版　　次：2018 年 5 月第 1 版　　**印　　次**：2018 年 5 月第1 次印刷
印　　数：1～1500
定　　价：29.00 元

产品编号：069486-01

内 容 简 介

本教材主要讲授 Python 程序设计知识，采用案例教学和问题驱动的撰写方法，注重实践思维、计算思维和创新思维等教育理念与教材内容的结合。本教材将知识点和实际应用相结合，以教学案例引出理论讲解。案例源于现实生活，旨在让读者理解实际问题被抽象化、模型化和程序化的全过程。内容涵盖 Python 应用的各个方面，以对比方式阐述人和计算机在解决问题时的异同，让读者理解计算思维的本质。教材在设计上由易到难，分别阐述计算机如何描述和处理现实世界中的各类事物，如何表示各类事物之间的关系，如何组织和优化程序结构等，使读者能够将程序设计和现实问题相关联。在讲解某一知识点时，横向延伸与之相关的各类知识点；在讲解某一个案例时，纵向扩展该案例所能实现的各种功能模块，使读者能够比较全面、深入地理解问题和掌握知识。教材穿插了一些技巧性、实用性的说明，并且对重要代码添加了注释。本教材免费提供与内容相配套的教学课件和各个案例的程序源代码。

本教材的内容涵盖范围较广，案例贴近实际，既可作为以 Python 为基础的程序设计类课程的配套教材，又可作为学习 Python 的很好的自学参考书，也适合各层次 Python 开发人员阅读参考。

图书在版编目(CIP)数据

Python 程序设计与实践：用计算思维解决问题/李莹主编；焦福菊，孙青编著. —北京：清华大学出版社，2018
ISBN 978-7-302-47389-3

Ⅰ. ①P…　Ⅱ. ①李…　②焦…　③孙…　Ⅲ. ①软件工具－程序设计　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2017)第 101920 号

责任编辑：张瑞庆
封面设计：何凤霞
责任校对：焦丽丽
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京鑫海金澳胶印有限公司
装 订 者：北京市密云县京文制本装订厂
经　　销：全国新华书店
开　　本：185mm×230mm　　**印　　张**：11　　**字　　数**：175 千字
版　　次：2018 年 5 月第 1 版　　**印　　次**：2018 年 5 月第1 次印刷
印　　数：1～1500
定　　价：29.00 元

产品编号：069486-01

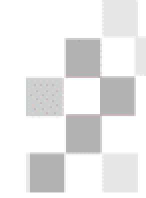


前 言

本教材主要讲授 Python 程序设计知识，包括良好的编程习惯、计算机描述现实事物、计算机处理现实事物、计算机的流程控制、计算机表示现实事物间关系以及程序编写方法共 6 章内容，涵盖了变量、数据类型、控制语句、数据结构、面向过程程序设计、面向对象程序设计、GUI 设计、网络编程和调试方法等重要知识点。本教材旨在让读者理解计算机解决问题的方法和思路，掌握程序设计的核心概念，构建基本的程序设计思想，学会编写中等难度的程序代码，为进一步学习和掌握计算机程序设计奠定良好的基础。

本教材采用“案例教学”和“问题驱动”的撰写方法，注重实践思维、计算思维和创新思维等教育理念与教材内容的结合。在编写时，按照问题求解的方式表述教学内容，不仅使学生掌握基本的程序设计知识，更重要的是教会学生解决问题的思维方法，即按照“提出问题、分析问题、讲解知识点、解决问题、总结思维方法”的思路组织教材内容。具体来讲，首先通过一个实例提出问题，然后分析解决问题的思路，引出解决该实例必须了解的核心概念和相关知识，并给出具体的解决方法，在完成具体的程序设计后，进一步展开阐述实用的编程技巧和工程实践经验，最后总结解决此类问题的思维方法，让学生不仅能够知其然，更能够知其所以然。撰写本教材的主要目标是学以致用，让学生掌握程序设计的基本技能，提高学生使用计算机解决实际问题的能力，同时更加注重学生计算思维和信息素养的培养，使他们具备用计算机抽象、分解、模拟和求解问题的能力，以及具备通过网络获取、分析和利用信息的自学能力。本教材的内容涵盖范围较广，案例贴近实际，既可作为以 Python 为基础的程序设计类课程的配套教材，又可作为学习 Python 的很好的自学参考书，也适合各层次 Python 开发人员阅读参考。

本教材的编写本着“案例引导知识、实践引导理论”的原则，将枯燥晦涩的理论性、原



理性的知识讲解转化成以问题驱动的案例教学。本教材设计和开发了一系列具有工程性、实践性、综合性等特点的教学案例。这些案例既联系所讲授的知识点，又注重学习者的学习兴趣，极大地激发了读者探究问题的求知欲。

本教材的主要特色和创新点如下：

(1) 案例丰富、贴近实际：将知识点和实际应用相结合，以教学案例引出理论讲解。案例源于现实生活，旨在让读者理解实际问题被抽象化、模型化和程序化的全过程。

(2) 内容全面、讲解独特：涵盖 Python 应用的各方面，以对比方式阐述人和计算机在解决问题时的异同，让读者理解计算思维的本质。

(3) 结构合理、设计新颖：教材以“用计算机解决现实问题”为主旨，在设计上由易到难，分别阐述计算机如何描述和处理现实世界中的各类事物，如何表示各类事物之间的关系、如何组织和优化程序结构等，使读者能够将程序设计和现实问题相关联。

(4) 难易适度、层层递进：教材采用横向和纵向两种方法撰写内容，在讲解某一知识点时，横向延伸与之相关的各类知识点；在讲解某一个案例时，纵向扩展该案例所能实现的各种功能模块，使读者能够比较全面、深入地理解问题和掌握知识。

(5) 代码注释、相关说明：为了便于读者的阅读和实现，教材穿插了一些技巧性、实用性的说明，并且对重要代码添加了注释。

(6) 配套课件、案例源码：教材提供与内容相配套的教学课件和各个案例的程序源代码。

参与本教材编写的都是北京航空航天大学计算机学院从事计算机基础教学多年、有着丰富教学经验的老师。其中，第 1 章和第 4 章由焦福菊、李莹编写，第 2 章、第 3 章和第 5 章由李莹、孙青编写，第 6 章由孙青、李莹编写。全书由李莹统稿并定稿。此外，在本书的编写过程中得到了李宇川的极大帮助，并且参考了国内外许多同类的优秀教材，在此表示深深的谢意。

由于时间仓促，加之编者水平有限，所以尽管经过了多次反复修正，但书中仍难免会有疏漏和不足之处，恳请同行专家、一线教师及广大读者批评指正。

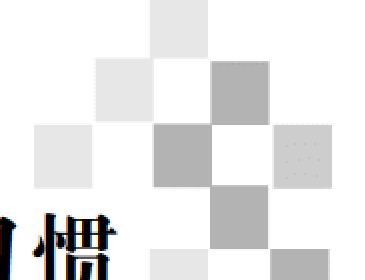
李　莹

2017 年 12 月于北京



目　录





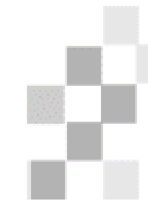
第 1 章　良好的编程习惯

随着互联网的快速发展，计算机的应用已经遍布社会生活的各个领域，并逐渐改变着人们的生产和生活方式。现代社会，每个人都应该学会使用计算机。

今天，我们随处可见，快递小哥使用智能手机接外卖订单，餐厅服务员使用终端为顾客点餐，出租司机使用智能手机抢单拉活，学生使用数字图书馆浏览图书资料。然而，人们使用计算机做事的层次是不同的。快递小哥、餐厅服务员、出租司机、学生等大多数人都是普通用户，他们只需要知道有哪些应用程序可以为他们做事，学习如何使用这些程序即可；而专业技术人员则应该学习如何编写和优化能够解决实际问题的程序。

无论是进行卫星轨道、天气预报等复杂计算的超级计算机，便捷使用互联网的智能手机，还是控制冰箱、洗衣机的嵌入式计算机，对于使用者来说，它们都只是一个能够接收指令并输出计算结果的机器，如何进行计算的步骤则需要人们通过程序来告诉计算机。计算机程序就是人们告诉计算机如何完成预定任务的步骤。

计算机只认识 0 和 1 两个数字，我们如何告诉它怎样去完成任务呢？这就需要学习如何用计算机语言来编写程序。计算机语言种类繁多、各有特色，对于不同类型的应用程序，可能用某种语言编写程序更加方便或运行效率更高。例如，开发网站可以用 PHP，控制网络数据传输可以用 C 语言，操作向量和矩阵可以用 MATLAB，等等。每种语言都有基本的程序结构、语法和语义。例如，输入输出、基本的数学和逻辑运算、有条件地执行、重复执行，等等。学会了使用一门计算机语言编程，再学习其他的语言就会容易得多了。



Python 是一门功能强大、简单易学的通用高级编程语言，非常适用于计算机程序设计的教学和计算思维的训练。本书将带你学会使用 Python 语言来分析和解决实际问题，并在学习和实践中养成良好的编程习惯。

1.1 Python 简介

Python 语言由荷兰人 Guido van Rossum 于 20 世纪 80 年代发明，它是一种动态的解释型语言，具有面向对象特征。由于其语法简明易学、代码可读性高、程序清晰美观、可移植性强等优点，越来越受到编程者的欢迎。Python 语言具有如下一些特点。

（1）**自由软件**：Python 是免费而且开放源代码的程序设计语言，它遵循 GPL（GNU General Public License）协议，谁都可以自由地发布这个软件的拷贝，也可以阅读和改动它的源代码，并将它的一部分应用到其他自由软件中。

（2）**简单易学**：语言本身的组成成分较少，结构较小。提供交互式环境，对于学习编程的新手而言，Python 提供的实时反馈非常有帮助。

（3）**解释型语言**：Python 拥有自己的解释器，不用编译、链接等源代码到机器代码的转换过程。把 Python 程序复制到另外一台机器上，Python 可以直接从源代码执行程序。

（4）**程序可读性高**：Python 更接近于自然语言，易于阅读。例如，变量类型不用预先定义就可使用，它的代码的外观与内在语义紧密相关，有利于初学者一开始就养成良好的编程习惯，非常适合于教学。

（5）**面向对象**：Python 不仅支持面向过程编程，还支持面向对象编程。它是一种公共域的面向对象的动态语言。

（6）**可扩展性好**：在 Python 脚本中可以嵌入如 C/C++ 等其他语言编写的程序，也可以将 Python 脚本嵌入到 C/C++ 等其他语言编写的程序中。

（7）**可移植性好**：由于 Python 的开源本质，Python 已经被移植到如 Windows、Linux、FreeBSD、Macintosh、VxWorks、Windows CE 等很多操作系统平台上。如果程序



谨慎地使用依赖于系统的特性，则所有的 Python 程序都无须修改就可以在上述平台运行。

(8) **丰富的库**：Python 拥有丰富的标准库以支持各种功能应用程序的开发。例如，文档生成、线程、数据库、网页浏览器、电子邮件、文件传输、网络接口、图形界面等有关的操作。除了标准库以外还有很多库，例如，wxPython、Twisted 和图像库等。

由于上述特点，Python 越来越多地被用作初学者的入门编程语言。本书所有的代码都是在 Windows 64 位操作系统下安装的 Python 3.4.0 环境中运行通过的。

说明：Python 是一种动态、解释型语言。

(1) **动态语言和静态语言**：动态语言是指在程序运行时确定数据类型的语言。变量使用之前不需要类型声明，通常变量的数据类型是被赋值的数据的类型。静态语言是指在编译时由变量的数据类型即可确定的语言，多数静态类型语言要求在使用变量前必须显式地声明其数据类型。

对于动态语言，变量可以在程序的不同位置被赋予具有不同数据类型的数据(数值型、字符串型等)；而对于静态语言，一旦变量被指定了某个数据类型，如果不经过强制类型转换，它将永远保持这个数据类型。

(2) **解释型语言和编译型语言**：解释型语言是在程序运行时，由与语言配套的解释器将程序逐条翻译成机器语言，即边解释边执行。解释型语言每执行一次就要翻译一次，效率比较低。编译型语言是在程序运行前，由与操作系统配套的编译器将程序整体翻译成机器语言，即一次编译、任意执行。编译型语言只需要在运行前完成一次翻译(原程序有变动除外)，而在运行时不需要翻译，程序的执行效率较高。

编译型语言与解释型语言，两者各有利弊。编译型语言由于程序执行速度快，同等条件下对系统要求较低，因此像开发操作系统、大型应用程序、数据库系统时都采用编译型语言，像 C/C++ 等语言基本都可视为编译型语言；而一些像网页脚本、服务器脚本及辅助开发接口这样的对速度要求不高、对不同系统平台间的兼容性有一定要求的程序，则通常使用解释型语言，如 Java、JavaScript、Python 等都是解释型语言。

1.2　Python 安装

1. Python 的安装

Python 语言的官方网站 www.python.org 为人们提供了免费的安装程序和学习文档。由于 Python 可以运行在多种操作系统平台，在下载 Python 时要注意自己的计算机所使用的操作系统和处理器的位数。例如，计算机是 64 位、运行 Windows 系统，则下载的文件应该是如图 1-1 方框中的 Windows x86-64 MSI installer。

Files

Version	Operating System	Description	MD5 Sum	File Size
Gzipped source tarball	Source release		e80a0c1c71763ff6b5a81f8cc9bb3d50	1943516
XZ compressed source tarball	Source release		8d526b7128affed5fbe72ceac8d2fc63	1430762
Mac OS X 32-bit i386/PPC installer	Mac OS X	for Mac OS X 10.5 and later	8491d013826252228ffcdeda0d9348d6	2482904
Mac OS X 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	349c61e374f6aeb44ca85481ee14d2f5	2317013
Windows debug information files	Windows		d6ffcb8cdabd93ed7f2feff661816511	3774378
Windows debug information files for 64-bit binaries	Windows		a0eea5b3742954c1ed02bddf30d07101	2503853
Windows help file	Windows		5fa4e75dd4edc25e33e56f3c7486cd15	7461732
Windows x86-64 MSI installer	Windows	for AMD64/EM64T/x64, not Itanium processors	963f67116935447fad73e09cc561c713	260546
Windows x86 MSI installer	Windows		e96268f7042d2a3d14f7e23b2535738b	249323

图 1-1　Python 文件下载

Windows 64 位操作系统下安装的过程很简单，按照提示安装即可。安装完成后，生成两种 Python 运行环境，一种是命令行方式，另一种是 IDLE 方式。IDLE 是一个综合编辑、运行、调试的集成开发环境，如图 1-2 所示，本书后面的实例都使用 IDLE 方式。

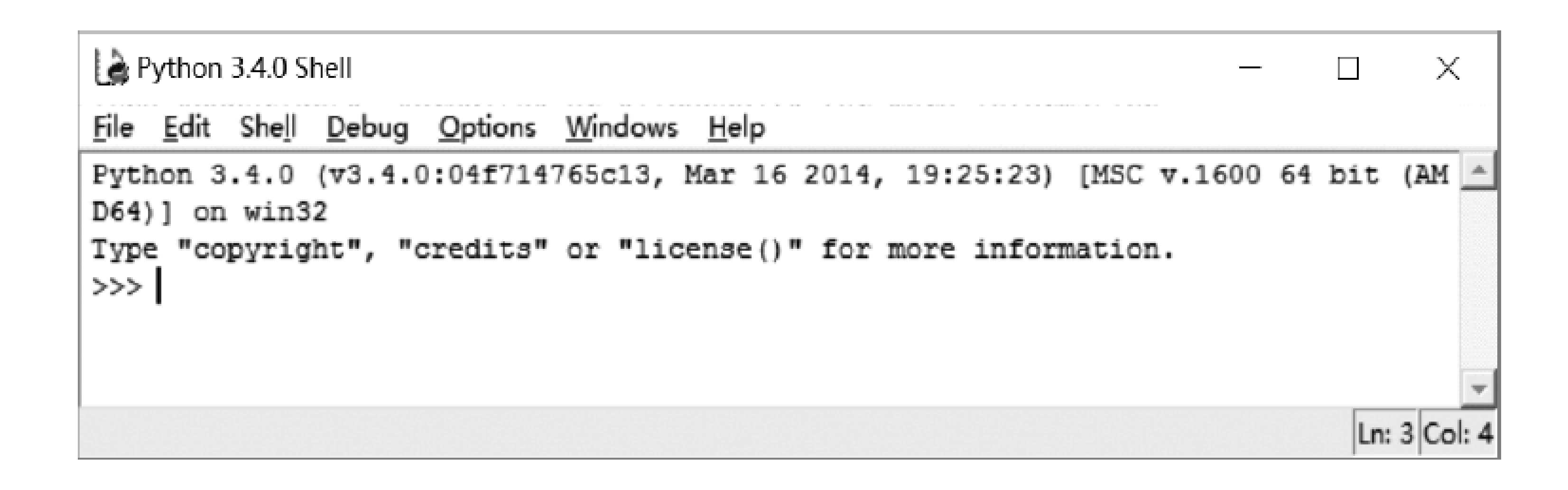


图 1-2　IDLE 集成开发环境

注意：尽量将 Python 安装在全英文路径下。

2. Python 初体验

现在从最简单的输出字符串“hello world!”入手来体验 Python 语言的魅力。

Python 的程序代码又称为脚本，是一系列指令的集合。Python 是动态、解释型语言，因此它拥有自己的解释器，每一条 Python 指令都可以直接执行。在 IDLE 集成开发环境中(Shell 窗口)输入一条指令，按回车键后，它会立刻显示结果。图 1-3 是 Python 3.4.0 命令行窗口，“>>>”是 Python Shell 的提示符。在提示符后面输入如下的指令：

```
>>>print("hello world!")
```

Python 立即将要打印的字符输出在屏幕上。print()是一个内置函数，输出指定的变量、函数或字符串的值。

```
Python 3.4.0 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("hello world!")
hello world!
>>> |
Ln: 5 Col: 4
```

图 1-3　Python 3.4.0 命令行窗口

有时可以像使用计算器一样执行一些简单的运算。例如，图 1-4 所示的语句，第一条语句将数值 10 赋给变量 a，第二条语句将数值 100 赋给变量 b，第三条语句计算 a 加 b 的

```
Python 3.4.4 Shell
File Edit Shell Debug Options Window Help
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 20:20:57) [MSC v.1600 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> a = 10
>>> b = 100
>>> a + b
110
>>>
Ln: 7 Col: 4
```

图 1-4　简单的 Python 语句

和。执行这三条指令后,Python 输出 a+b 的结果。

输入下面的指令会输出什么结果呢?请读者自行练习。

```
>>>21 * 30
>>>5+100.30
>>>1+9j+3
>>>2+8j * 3
>>> (5+3j) * 3
```

从上面几个实例可以看出,Python 不仅支持整数运算、浮点数运算,还可以支持复数运算。加、减、乘、除等运算符号是有优先级的,括号可以改变运算符的优先级。有关数据类型的各种操作将在第 2 章中讨论。

再输入下面的指令,看看又会输出什么结果。

```
>>>s="hello"
>>>t="world  ! "
>>>print(s+t)
>>>a=    'Is it a cat? '
>>>b=" 'No,it's not.' she said."
>>>print(a+b)
>>>c=100
>>>d="123"
>>>print(c+d)
```

这些实例说明 Python 不仅可以处理数字,也可以处理字符串。在 Python 中,字符串是由双引号或单引号括起来的字符组成的,这里的字符包括字母、数字和控制字符。如果字符串本身包含单引号,则上面的语句“b=" 'No,it's not.' she said." ”是正确的,而写成“b='"No,it's not." she said.'”则会报错。这时,我们也可以用转义字符“\”来表示中间的单引号,例如“'"yes,it\'s " she said.'”。字符串类型的数据也可以做连接、截取等操作,但字符串和数值是不能直接做算术运算的,有关字符型数据的操作将在第 2 章中讨论。

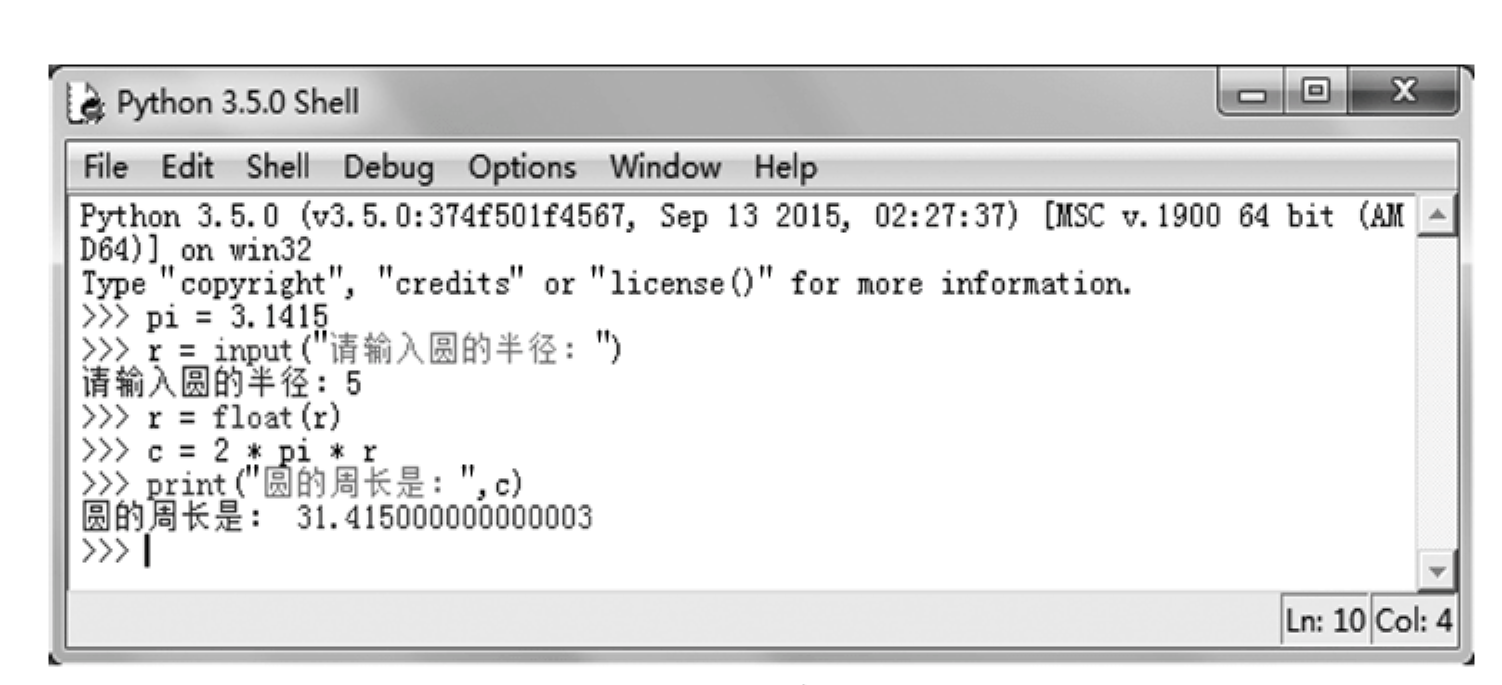
3. 编写 Python 程序

Python 提供两种编写程序的方式：

（1）Shell 窗口编写；

（2）文件窗口编写。

Shell 窗口提供一种交互式的编程模式，它一般用于编写简单的程序，如图 1-5(a)所示。Shell 窗口的特点是“边输入指令、边执行并输出结果”，即输入的每条 Python 指令会在按下 Enter 键后被立即执行。只要不打开新的 Shell 窗口(IDLE)，前面定义的变量在后面的指令中都可以使用。一旦关闭 Shell 窗口，会话中的所有变量和输入的语句就不存在了。为了使程序代码能够被重复执行，需要将代码保存到文件中。和其他编程语言一样，我们需要用文件窗口来编写和保存源代码，如图 1-5(b)所示。文件窗口的特点是“输入完整代码后一次执行并输出结果”。Python 源代码文件是普通的文本文件(*.py)，可以用任意能够编辑文本文件的编辑器来编写 Python 程序，如记事本、Word 等。



(a) Shell窗口

circle.py - C:/Users/new/Desktop/python/circle.py (3.5.0)

File Edit Format Run Options Window Help

```
#求圆的周长
pi = 3.1415
r = input("请输入圆的半径：")
r = float(r)
c = 2 * pi * r
print("圆的周长是：",c)
```

Ln: 5 Col: 14

(b) 文件窗口

图 1-5　Python 集成开发环境

在IDLE中单击File|New File菜单，在打开的可编辑环境中输入Python语句，并保存为*.py格式文件，这样就可以实现程序的反复查看、修改和重复执行了。

另外，Python的IDLE还有内容的高亮显示功能，即可以根据输入内容(如程序注释、关键字等)的不同而自动识别并显示不同的颜色，使得程序显得更加清晰、易读。本书后面的实例如果没有特别说明，则是在文件中编辑、编译和执行的。

1.3 漂亮的程序

程序是一件艺术品，一个符合规范的程序是"十分漂亮的"。这里，"漂亮"有以下两层含义：

(1) 满足编程语言的语法规则：在Python中，体现代码层次关系的缩进(4个空格)和冒号":"都是语法规则，不能省略。

(2) 符合阅读程序的审美习惯：编程时，为了提高程序的可读性和可维护性，通常会对关键语句添加注释(以#开头的语句)，也会在不同代码块间增加空行。这些操作不属于Python的语法规则，虽不是必需的，但却是常用的。

可见，养成规范的编程习惯，对于一个程序员来说是非常重要的。下面试着读一读例1-1给出的程序代码。

【例1-1】 计算两个数的最大公约数。

对比下面的两个程序。

(1) 无层次区分的程序

```
a=input("please input the first number:")
b=input("please input the second number:")
num1=int(a)
num2=int(b)
if num1<num2:
temp=num1
num1=num2
```

```
num2=temp
while num2 !=0:
temp=num1 %num2
num1=num2
num2=temp
print(a,"和",b,"的最大公约数是:",num1)
```

(2) 有层次区分的程序

```
a=input("please input the first number:")
b=input("please input the second number:")
num1=int(a)
num2=int(b)

if num1 <num2:
    temp=num1
    num1=num2
    num2=temp

while num2 !=0:
    temp=num1 %num2
    num1=num2
    num2=temp

print(a,"和",b,"的最大公约数是:",num1)
```

先不用急于读懂代码,仅从形式上对比上面的两个程序,就能直观地感受到有层次区分的程序在代码可读性、结构逻辑性等方面的优势。另外,无层次区分的程序不但难以理解,更重要的是其执行时会报错。究其原因,是因为它不符合 Python 的编写规范(详见 1.3.1 节)。

说明:(1) 空格的使用:空格常被用来修饰程序,使其看起来更加漂亮且易于阅读。一般会在二元操作符两边。例如,赋值操作符(=),比较操作符(==、<、>、!=、<>、

```
num2=temp
while num2 !=0:
temp=num1 %num2
num1=num2
num2=temp
print(a,"和",b,"的最大公约数是:",num1)
```

(2) 有层次区分的程序

```
a=input("please input the first number:")
b=input("please input the second number:")
num1=int(a)
num2=int(b)

if num1 <num2:
    temp=num1
    num1=num2
    num2=temp

while num2 !=0:
    temp=num1 %num2
    num1=num2
    num2=temp

print(a,"和",b,"的最大公约数是:",num1)
```

先不用急于读懂代码,仅从形式上对比上面的两个程序,就能直观地感受到有层次区分的程序在代码可读性、结构逻辑性等方面的优势。另外,无层次区分的程序不但难以理解,更重要的是其执行时会报错。究其原因,是因为它不符合 Python 的编写规范(详见 1.3.1 节)。

说明:(1) 空格的使用:空格常被用来修饰程序,使其看起来更加漂亮且易于阅读。一般会在二元操作符两边。例如,赋值操作符(=),比较操作符(==、<、>、!=、<>、

<=、>=)，布尔操作符(and、or 和 not)等符号的两边都加上一个空格,但括号内一般不会在操作符两边添加空格。

(2) 空行的使用：通常会在具有相对独立功能的代码块间(如数据输入、数据处理、结果输出等)添加空行,从而使程序结构清晰,便于理解。

1.3.1 语法规则

1. 常量和变量

数据是程序处理的对象,是构成程序的最基本要素。因此,编写程序首先要考虑如何表示数据。数据可以被细分为常量和变量。常量是不能够被改变的量,例如,"hello world!"、150.25、True、False 等;而变量是它的值可以改变的量,Python 通过变量名来区分不同的变量,且变量名区分大小写,例如,“sum=3”就是给变量 sum 赋值为数值类型的 3,且它的值和类型都可以根据需要而被修改。

2. 表达式

按照算法的要求用运算符将常量、变量等组合起来就形成了表达式。例如,表达式(a+b+c)/2 表示将 a、b、c 三个变量的值相加,然后除以 2;而表达式"he is"+"joy"表示将字符串"he is"与字符串"joy"连接起来。

3. 语句

语句是构成计算机程序的基本单元,Python 提供多种语句类型,每种语句类型都有自己的语法规则,并且实现不同的功能。最常用的是赋值语句,它实现给变量赋值的功能,等号“=”是赋值运算符,表示将右边的值赋给左边的变量。例如:

```
>>>num=100
>>>name="John"
```

Python 是解释型语言,因此,变量类型是由赋给它的数据值直接决定的,不需要事先声明。例如,num 被赋值为 100,则它的数据类型就是数值型,而 name 被赋予字符串"John",则它的数据类型是字符串型。具有不同数据类型的变量可以执行相同的运算,



但是具有不同的含义。例如,1＋2 和"I"＋"am"。注意,具有不同数据类型的变量之间不能直接进行运算。例如,"number"＋100 则会报错。

【例 1-2】 简单的猜数游戏。

```
a=input("猜猜我是几:")
a=int(a) #强制类型转换
b=57
if a==b:
    print("对了!")
else:
    print("不是哦!")
```

从形式上可以看出,if 语句后必须要有冒号“:”,且 if 中的各条语句都需要空 4 个空格以区分不同的层次结构。

if 语句用来实现程序的分支结构(详见第 4 章)。例 1-2 的语句表示:如果变量 a 的值等于 b 的值,则输出结果“对了!”;否则,输出结果“不是哦!”。

说明:(1) input([prompt])语句是 Python 3 提供的输入函数。它用于接收从控制台上输入的数据,返回为字符串类型(string 类型)。参数 prompt 是字符串类型,用于在控制台上输出指定的提示信息。Ctrl＋Z 结束输入。例如,num＝input("请输入第一个数:\n")语句,表示实现将用户在键盘上输入的内容赋给变量 num,括号中的字符串是显示在屏幕上的提示信息,\n 为提示信息换行。

注意,input()函数返回的结果是字符串类型,即 num 被赋值为字符串类型的变量。如果希望得到数值类型,必须进行强制类型转换,即 num＝int(num)。

(2) print()语句是 Python 3 提供的打印函数。它的参数比较丰富,用于指定输出一个或多个表达式的值。例如,print(num)语句表示输出变量 num 的值,print("The number is:",num)语句则表示在同一行输出字符串"The number is:"和变量 num 的值。

更多用法,请读者自行查阅 Python 语言的相关手册。

【例 1-3】 计算 1～10000 的累加和。

```
i=1
sum=0
while i<=10000:
    sum=sum+i
    i=i+1
print("sum=",sum)
```

从形式上可以看出，while 语句后必须要有冒号“:”，且 while 中的各条语句都需要空 4 个空格以区分不同的层次结构。

while 语句用于实现代码的重复执行(详见第 4 章)。例 1-3 中的语句表示：计算 1～10000 的累加和。其中，i 为循环控制变量，用于控制循环次数。当 i 小于或等于 10000 时，重复执行“sum＝sum＋1”和“i＝i＋1”两条语句，直到“i＜＝10000”的判断条件不满足时，程序才退出循环，并继续执行后面的 print()语句。

通过上面两个例子可以得出，“缩进”(4 个空格)和“冒号”都是 Python 程序中的语法规则，必须严格遵守，否则报错。

4. 程序的语法规则

编写 Python 程序时，请记住几个基本的语法规则：缩进、冒号、空行。

(1) 缩进：缩进是 Python 的一种语法规则，具有特殊含义。Python 用行首前的 4 个空格来表示行与行间的层次关系。代码缩进一般用在 if、while 等控制语句和函数定义、类定义等语句中。例 1-3 的 while 循环语句中，“sum＝sum＋i”和“i＝i＋1”这两条语句是 while 语句的循环体，所以这两条语句前必须加入 4 个空格进行缩进。而后面的 print()语句不属于 while 语句，所以不需要缩进。

(2) 冒号：冒号是 Python 的一种语法规则，具有特殊的含义。在 Python 中，冒号和缩进通常配合使用，用来区分语句之间的层次关系。例如，在 if 和 while 等控制语句以及函数定义、类定义等语句后面要紧跟冒号“:”，然后在新的一行中缩进 4 个空格，输入语句主体。例 1-2 中的 if 语句，在条件判断表达式“a＝＝b”的后面必须有“:”，然后再输入

满足该条件的语句。

(3) 空行：空行不是 Python 的一种语法规则。当存在多个函数、类定义或相对独立的代码块时，函数间、类间或代码块间常用空行分隔，使得程序更加清晰、易读。例 1-5 中，程序定义了两个函数，分别实现打印三角形和倒三角形的功能。程序中用若干个空行将两个函数的定义和对它们的调用分隔，使得结构清晰、层次分明。

另外，缩进是可以嵌套的，缩进的层次不同，则语句间的从属关系不同。

【例 1-4】 修改例 1-2，使程序不仅能判断是否猜测正确，还能判断输入数据和被猜数据间的大小。

```
a=input("猜猜我是几：")
a=int(a) #强制类型转换
b=57
if a==b:
    print("对了！")
else:
    if a>b:
        print("猜高了！")
    else:
        print("猜低了！")
```

【例 1-5】 打印用字符组成的菱形。

```
def  triangle1(): #打印正三角形的函数定义
    print("    T")
    print("   TTT")
    print("  TTTTT")
#空行
def  triangle2(): #打印倒三角形的函数定义
    print("  TTTTT")
    print("   TTT")
    print("    T")
```

```
#空行
triangle1() #调用打印正三角形的函数
triangle2() #调用打印倒三角形的函数
```

1.3.2 注释规范

注释用于在程序中解释变量的意义、说明函数的功能、标注程序模块的创建者和创建模块的时间等,以便帮助编程者和阅读者能够更好地理解程序。养成为程序添加注释信息的好习惯,既方便自己以后修改程序的功能,又方便与他人合作开发软件。

Python 中有以下两种添加注释的方式。

(1) 单行注释:以"#"开头的一行信息。

(2) 多行注释:以一对三引号(包括三个单引号或三个双引号)"'''"或""""""包含的多行信息)。

在程序中,注释语句不会被解释器解释和执行。

【例 1-6】 给例 1-1 中的代码添加注释。

```
#计算两个数的最大公约数
a=input("please input the first number:")
b=input("please input the second number:")
num1=int(a) #对 a 进行强制类型转换
num2=int(b)

if num1<num2: #如果 a 小于 b,则互换 a 和 b
        temp=num1
        num1=num2
        num2=temp
#欧几里得算法计算两个数的最大公约数
while num2!=0:
        temp=num1 % num2
        num1=num2
```



```
        num2=temp

print(a,"和",b,"的最大公约数是:",num1)
```

与例 1-1 相比，例 1-6 的代码更容易理解和阅读。

通过上述简单的程序，我们了解到：在 Python 中，语句的缩进和冒号十分重要，注释使程序结构更加清晰、内容更加易懂。Python 将缩进作为语法规则，就是要强制初学者养成良好的编程习惯。一个好的程序，不仅要有正确的算法，还要有清晰的逻辑结构和简明扼要的说明和注释。

1.3.3　程序调试

了解了 Python 的基本语法规则后，就可以开始编写一些简单的程序了。再简单的程序也难免会出错。一般而言，每编写 1000 行代码就可能有 3～10 条错误语句。这些错误主要包括语法错误、运行错误和逻辑错误。所谓“知错就改就是好程序”。出错不可怕，最关键的是定位出现错误的位置、分析产生错误的原因以及掌握修正错误的方法。

不同的错误类型，可以采用不同的解决方法。当出现语法错误或运行错误时，可能会导致程序崩溃，Python 解释器将抛出错误产生的位置和错误类型等异常信息。这些信息对错误的查找和修改起着至关重要的作用。因此，这两类错误相对容易解决；而当出现逻辑错误时，程序一般能正常执行，Python 解释器没有检查或捕获到任何错误，但程序运行的结果却不是我们想要的。逻辑错误意味着算法在设计上可能出现了逻辑问题，由于没有解释器的错误提示，查找程序的逻辑错误相对要困难得多，这就需要从头至尾逐条地审查程序代码。一种常用的帮助排除逻辑错误的方法是在程序中插入 print()或 assert()等插桩语句，用来输出中间结果，通过对中间结果的观察可以精确地定位错误。一旦错误被修正后，再删除或注释掉中间插入的 print()或 assert()等插桩语句。

调试程序是一个集知识、经验、直觉和耐心于一体的过程，需要初学者不断总结引发错误的原因以及解决错误的方法。

下面列举一些常见的语法错误或运行错误。

(1) 程序中的语句或表达式出现操作数不完整、使用没有赋值的变量名、缺少冒号“:”或者没有缩进等错误。例如:

```
>>>299+33*
SyntaxError:invalid syntax
```

错误信息表明输入的语句是一个无效的表达式,显然乘号“*”后面还需要有另一个操作数。又如:

```
>>>a+1
Traceback (most recent call last):
File "<pyshell#1>", line 1, in <module>
a+1
NameError: name 'a' is not defined
```

错误信息表明变量 a 没有定义,即输入语句中的变量 a 没有被定义过,不能直接使用。再如:

```
>>>while i==1
    print("hello world! ")
SyntaxError: invalid syntax
```

错误信息表明是语法错误,导致错误的原因是 while 的条件判断语句“i==1”后缺少冒号“:”以及 while 内部的“print("hello world! ")”语句没有缩进。

(2) 程序执行过程出现数据类型不匹配、除数为 0、函数的参数数量或类型不一致等错误。例如:

```
>>>'9'+90
TypeError: Can't convert 'int' object to str implicitly
```

错误提示表明 Python 无法将 int 类型自动转换为 str 类型,即输入语句直接将字符类型和整型一起计算,从而导致了数据类型不匹配的错误。再如:

```
>>>i=0
```

```
>>>100 * (100/i)
ZeroDivisionError: division by zero
```

错误提示表明除数为 0。

提示：调试程序时，一般都从 Python 编辑器抛出的最后一条错误信息开始定位错误和查找原因。这是因为，前面抛出的若干条错误信息很可能都是由后面的错误导致的。

1.4　Python 学习资料

随着 Python 语言的广泛流行，网络上 Python 编程的学习社区和网站也大量出现。这里按照 Python 安装包下载、Python 第三方库文件、Python 技术文档、Python 视频课程等分门别类地列在下面。

1. Python 安装包

Python 官方安装包下载的网址为 www.python.org。

2. Python 第三方库文件

Python 提供三种安装第三方库文件的方法：

(1) 通过 setuptools 来安装 Python 模块。

(2) 通过 pip 来安装 Python 模块。

(3) 从网上下载可执行文件直接安装。

第三方开发模块下载的网址主要有 http://www.lfd.uci.edu/～gohlke/pythonlibs/和 pypi.python.org。

3. Python 技术文档

Python 官方文档中文版网址为 http://python.usyiyi.cn/。

4. Python 学习社区

(1) Python 中文开发者社区，按照基础、高级、框架、函数等分类提供学习资料，论坛和在线学习手册非常实用，网址为 http://www.pythontab.com/。

(2) 玩蛇网 Python 学习与分享平台，提供 Python 教程讲解、实例源码、各种应用编程等，如果深入学习需要报班，网址为 http://www.iplaypython.com/。

(3) Python 中国，是 Python 开源分享社区，网址为 http://www.okpython.com/。

5. Python 视频教程

(1) 麦子学院，网址为 http://www.maiziedu.com/course/python/。

(2) 极客学院，网址为 http://e.jikexueyuan.com/python.html。

(3) 爱酷学 Python 在线学习视频，网址为 http://www.icoolxue.com/album/affix/view/python/1/8? orderBy=create_time。

习　题

1. 在 Python 命令行窗口输入以下语句，请预测输出结果，若无输出结果，请说明错误原因。

(1) 3+4

(2) 1+2.23

(3) 1+2j*3

(4) "3"+4

(5) "3"+"4"

(6) 1+2+3+n

(7) 3+4(5+6)

(8) 10/0

2. 简述常量与变量的区别。

3. 简述表达式与语句的区别。

4. 注释在程序中起什么作用？如何表示注释？

5. 在文件中输入并运行下列程序，指出错误类型及原因。

(1) 计算两个数的商。

```
a=input("please input the first number:")
b=input("please input the second number:")

c=a/b
print("a除以b等于:",c)
```

(2) 判断两个数是否相等。

```
a=int(input("please input the first number:"))
b=int(input("please input the second number:"))
if a==b
print("a和b相等")
```

(3) 计算 1～10 的累加和。(提示：有两个错误)

```
i=1
while i<10:
    my_sum=my_sum+i
    i=i+1
print("1~10的累加和为:",my_sum)
```



第2章　计算机描述现实事物

程序是由“输入、处理和输出”三要素组成的。其中，输入：要解决问题的抽象描述；处理：用算法解决问题；输出：显示结果。所以，学习编程，首先要学会如何使用计算机描述现实世界的各类事物，即将客观事物映射并存储到计算机内存中，以便程序的读写和处理。计算机的一维一元存储特性和人固有的多维多元的信息获取方式截然不同，这给表示和存储客观世界的一切事物和它们之间的复杂关系带来了极大障碍。因此，使用计算机解决问题的首要任务就是将人脑中的多维多元信息转换为计算机中的一维一元信息，然后被直接或间接地映射到连续或非连续的内存线性空间中。计算思维中的“降维思维”起了关键性作用。降维思维是将人脑丰富多彩的高维度思维空间，逐步转换、映射降维到较低维度的思维空间，直至线性空间，使得复杂问题逐步化简，最终实现机器可解的思维过程。在降维过程中，要考虑维度变化时信息量的损失和表达的误差。用降维思维可以表达现实世界的一切事物和现实世界的一切关系(详见第5章)。在表达现实世界中的各类事物时，根据事物的描述方式，可将其分为数值类型和非数值类型两大类。其中，数值类型又分为数字类型和布尔类型，这类事物的表达和存储方式和人脑类似，不需要进行降维转换，可以直接将人脑接收到的信息按二进制方式存储到计算机中。而对于像文本、图形、图像、视频和声音等非数值数据的计算机表达，就必须通过数字化手段进行降维处理，将人脑中复杂的信息存储形式转换为单一的数字形式。



2.1　变量

程序是为解决特定问题而用计算机语言编写的指令集合。它具有输入、处理和输出三个基本要素。其中,学会输入是写好程序的第一步。编程时,直接采用显式方式输入和存储数据本身是十分困难的。这不仅给编程人员带来记忆上的负担、书写上的繁琐,还使得程序不易被理解、修改和扩展。例如,计算圆的面积和周长时,用 3.1415926 表示圆周率,用 3 表示圆半径,则程序可写为:

```
print("The area of the circle is",3.1415926*3*3)
print("The circumference of the circle is",2*3.1415926*3)
```

如果想提高计算精度或计算不同圆的面积和周长,我们只能在程序中逐一查找和修改相应的数据,操作繁琐且容易出错。因此,我们化繁为简,给抽象难懂的数据起一个有意义的名字,称之为变量。变量是程序的重要概念,程序中对数据的各种操作都是通过变量实现的。

【例 2-1】　使用变量计算圆面积和周长。

```
r=3
my_pi=3.1415926
area=my_pi*r*r
cir=2*my_pi*r
print("The area of the circle is",area)
print("The circumference of the circle is",cir)
```

2.1.1　变量的含义

所谓变量,就是值可以改变的量。它源于数学,但是在计算机中,变量又有自己的一些特殊含义:

(1) 变量是内存中一段连续的存储空间,可以被理解为是一个容器,我们用它来暂存

CPU处理时需要的数据。存储空间的大小取决于变量的类型(详见2.2节)。变量名是该容器的名字,其实就是数据在内存中的地址,它告诉程序到内存的哪个地方读写数据,如图2-1所示。程序访问到某个变量,就会按名取值,即根据变量名读取其所存储的数据值,通常将变量名简称为变量。

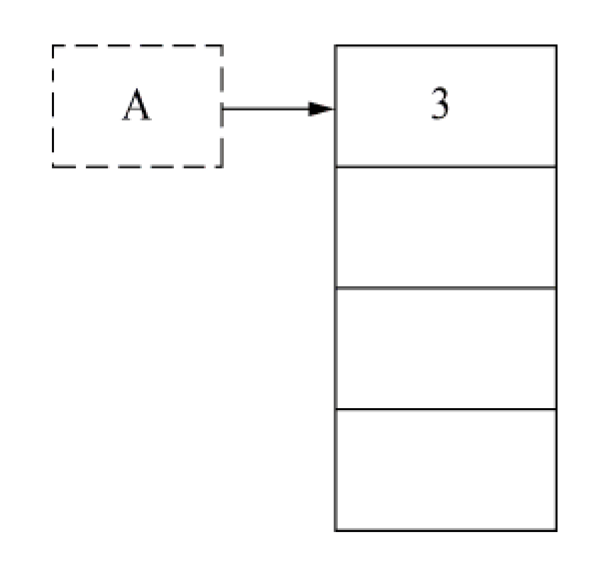


图2-1 内存中的变量名

提示: 内存是暂时存储程序和数据的地方,它是由许多存储单元组成的一维空间。我们可以将内存比喻成一座大楼,但每个楼层仅有一个房间,每个房间都有自己唯一的编号且各房间的大小相同(仅能容纳8个座位)。每个房间就是一个存储单元,房间大小就是存储单元的容量,房间编号就是存储单元的地址。存储单元的容量用来表示每个存储单元所能存放的二进制数的位数。现在常用的计算机的存储单元容量为8位(8位=1字节),即存储器是以字节为单元存储数据的;内存地址用于表示每个存储单元在内存中的位置。根据内存地址,我们可以正确地获取存储单元中的内容。编程时,可以用变量名代替内存地址来访问存储单元,从而降低程序编写的难度。这里,变量名可以视为内存地址的别名。因此,从本质上说,变量代表了一段用户可操作的内存空间,它是内存的符号化表示。

理解: 变量名和变量值是两个完全不同的概念。其中,变量名是指内存中存储数据的具体位置;而变量值是指变量名所指引的内存单元中的内容,即具体的数据值。

(2) 变量具有不同的类型,如数值类型、字符串类型等(详见2.2节)。变量的类型是由赋值给它的数据类型所决定的。所谓"物以类聚",不同类型的变量是不能直接进行计算的,就如3个苹果和2只兔子不能被放在一起计算一样。

提示: Python是动态语言,即变量类型是根据其所赋值的数据类型动态变化的。因此,Python不需要声明变量,即不需要在定义变量时显式地说明其类型。变量的类型是由赋值时的数据类型所决定的。例如,A=3,此时变量A是一个整型类型;B="hello",则变量B的类型是字符串类型。在Python中,可以用type()函数查看变量类型。

(3) 变量的值可以根据需要随时改变。在改变变量值的同时,变量的类型也会随之

改变。例如,A=3,此时变量 A 是一个整数类型,如图 2-2(a)所示。重新给 A 赋值,A="hello",则变量 A 的类型“变为”字符串类型,图 2-2(b)所示,实际上,并不是改变了变量 A 的类型,而是创建了一个新的变量,即整数类型的变量 A 和字符串类型的变量 A 是两个变量。另外,变量有“喜新厌旧”的特性,它只记住当前的值,而遗忘过去的值。

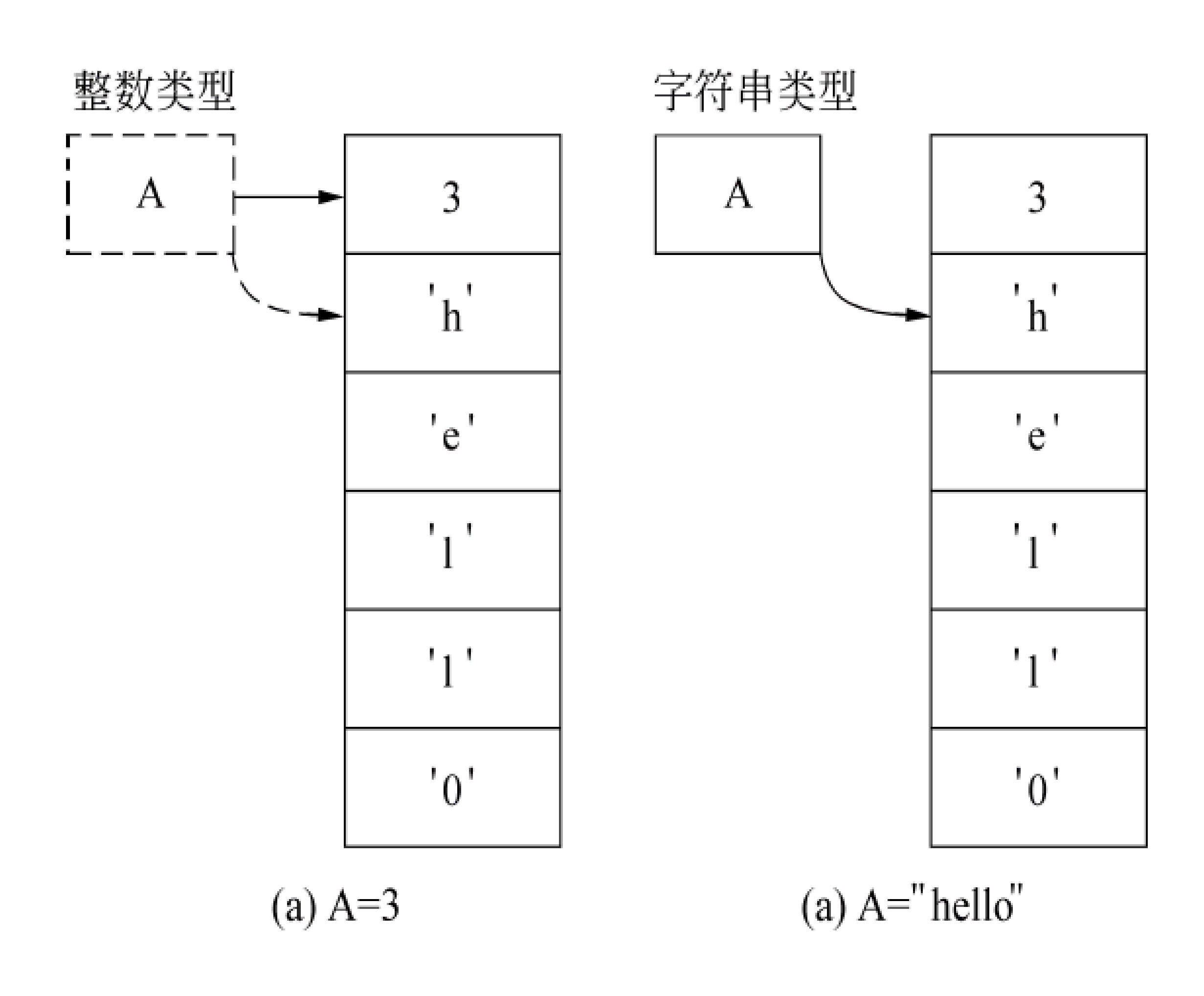


图 2-2　变量的类型与变量的值

说明:当改变变量的值时,并不是改变该变量所指的原存储单元中的内容,而是在内存中开辟一个新的空的存储单元,将新值写入该存储单元中,并将变量名重定向到该存储单元的地址,即 A=3 和 A="hello"分别表示两个存储单元的内容。因此,Python 是一种强类型语言,从本质上看,其变量的类型一经确定是不能再被改变的。在 Python 中,可以用 id()函数查看变量所在的内存地址。但 A=3 由于变量被赋新值而无法再被访问,会导致内存泄漏。所谓内存泄漏,就是内存中的数据一直占据存储单元却无法被访问,直到程序结束。因此,在改变变量的值之前,一定要判断其原值是否有用。如果有用,则需要将其赋值给其他变量;如果没用,则需要及时删除。在 Python 中,可以用 del()函数删除对象。

理解:变量可以被理解为是一个标签,A=3 表示将标签 A 贴在了一个存储内容为 3 的内存单元上;重新给 A 赋值,A="hello"表示将标签 A 从原内存单元撕下,贴到了一个新的存储内容为"hello"的内存单元上。

(4) 在 Python 中,变量名区分大小写,即 A=3 和 a=3 表示不同的变量;但是,一个数据可以有多个变量名,即同一个存储单元可以有多个别名。

提示:①不同的变量可能指向同一个存储单元(如 A=3,B=A),也可能指向不同的存储单元(如 A=3,C=4);②当不同变量指向不同的存储单元时,它们的值可能相同,也可能不同。例如,A=3 和 a=3 是两个不同的变量,它们指向不同的存储单元,但它们存储的数据值相同。

(5) 不同变量有不同的作用域,包括全局作用域和局部作用域。根据作用域的不同,可以将变量分为全局变量和局部变量。其中,全局变量的作用域最大,它能在程序的任何地方被访问;而局部变量的作用域较小,它仅能在其被定义的函数内部被访问。

(6) 不同变量有不同的生命期,包括长生命期和短生命期。其中,全局变量具有长生命期,它能在程序运行的整个过程中存活;而局部变量具有短生命期,它仅能在被定义的函数或模块运行期间存活。

说明:生命期是一个时间概念,表示变量生存的时间长短;而作用域是一个空间概念,表示变量起作用的空间范围。变量和人类似,有自己的存活期和行使权利的范围。这些都与其所处的"位置"有关。例如,市长一般比县长的管辖范围大;而医疗条件差的人普遍比医疗条件好的人的寿命要短。因此,变量的作用域和生命期与其在内存中的存储位置息息相关。这里,需要了解一些内存知识:根据存储内容的不同,可以将内存分为代码区、静态存储区、栈区和堆区4个区域。程序运行时,内存有三个区域可以保存变量,它们是静态存储区、栈区和堆区。另外,在特殊情况下,变量还可以被保存在CPU的寄存器中。其中,①静态存储区:存放程序中的全局变量、静态变量和常量。其特点是,该内存区域在程序编译时就被分配好,在程序运行的整个期间一直存在,在程序结束后,才会被释放。存储在该区域的变量具有长生命期,即其寿命和程序一样。②栈区:存放程序中的局部变量,即各函数中的变量。其特点是:该内存区域是在变量所在函数或模块被执行时由操作系统自动创建的。当函数或模块执行完毕后,该内存区域也会被自动释放。每当函数或模块被执行一次,其定义的局部变量就会被重新分配内存空间。③堆区:存放程序的动态变量。其特点是:该内存区域是在动态变量创建时由程序员手动创



建的。当动态变量使用完,该内存区域也应该由程序员手动释放。若程序员没有及时释放,则会在程序结束时由系统自动释放。④寄存器:CPU 中的寄存器数量非常有限且存储容量很小,但访问速度却很快。因此,它常被用于存储频繁的变量。

【例 2-2】 执行以下程序,并且写出各变量的值。

(1) 请写出 MyCtity 和 YourCity 的值。

```
MyCity="Beijing"
YourCity=MyCity
MyCtity="Shanghai"
```

(2) 请写出 MyAge 和 YourAge 的值。

```
MyAge=24
YourAge=24
YourAge=26
```

解答:

(1) 开始时,YourCity 和 MyCity 指向同一个内存单元,它们具有相同的变量值"Beijing"。但是将 MyCtity 的值修改为"Shanghai"后,系统并不是修改原有 MyCtity 所指向的内存单元的值,而是在内存中重新开辟了一个新的内存单元,填写其内容为"Shanghai",并将 MyCtity 重定位到该内存单元。

最终,MyCtity 的值为"Shanghai",YourCity 的值为"Beijing"。

(2) MyAge 和 YourAge 虽然值相同,但它们分别指向两个独立的内存单元,互相没有影响。因此,修改 YourAge 的值不会改变 MyAge 的值。

最终,MyAge 的值为 24,YourAge 的值为 26。

2.1.2　变量的命名

变量命名看似简单,却是让程序员最头痛的事情。2015 年 10 月,在 ITworld 发起的 Programmers' hardest tasks(程序员最头疼的事)的调查中,共有 4522 人参加投票,其中

约有 49%认为命名是最困难的。

可见,给变量起一个合适的名字就像给人起一个好名字一样,需要满足很多要求。Python 有自己的一套特殊的命名规则,包括对变量(局部变量、全局变量)、函数、类、模块、包等的命名。在 Python 中,对变量的命名必须遵循以下基本命名规则:

(1) 变量名必须以字母或下画线开头,如 1area 是错误的变量名。

(2) 除首字符外,变量名可以包含任何字母、数字和下画线的组合。

(3) 除下画线外,变量名中既不能出现其他任何特殊字符,如分隔符、空格、标点符号或运算符号(~、!、@、#、$、%、^、&、*、—、+等);也不能出现系统保留字(通常在 IDLE 中高亮显示)。

(4) 变量名长度不受限制。

(5) 变量名区分大小写。

(6) 变量名不能与 Python 自带的关键字重名,Python 中常用的关键字如表 2-1 所示。

另外,选取变量名有一个重要原则,就是"见名知义"。例如,用 sum 作为一个变量名来表示和,就比用 a 或 b 等来表示和清晰得多。良好的命名风格会使程序更加清晰易懂,在初学编程时,养成良好的编程习惯是非常必要的。本教材中比较简单的例子可能用到 a、b 等无意义的名字,而正式的程序还应该遵循一定的命名规则。

表 2-1　Python 中常用的关键字

and	as	assert	break	class	continue	def	del	elif	else	except	exec
False	finally	for	from	global	if	import	in	is	lambda	not	None
or	pass	print	raise	return	try	True	while	with	yield		

说明:下面介绍 Python 中的常用命名习惯。

(1) 编程时,一般会根据变量、函数、类、模块、包等所要表示的含义采用其英文单词或缩写进行命名。这里,介绍两种编程时常用的命名方法:下画线命名法和驼峰命名法。①下画线命名法就是用下画线区分变量名中的单词。对于局部变量名、模块名和包名等



一般采用全小写字母＋单下画线方式命名，如 student_name、numpy_creat_data；全局变量名一般采用全大写字母＋单下画线方式命名，如 COUNTER、STUDENT_NUMBER；类名一般采用首字母大写命名，如 Circle()、ClassName()。②驼峰命名法就是混合使用大小写字母来构成变量和函数的名字，一般以小写字母开头的一个或多个英文单词作为变量名，如 studentName。

(2) 双下画线一般用于类中定义的各种变量，其他地方不建议使用。

(3) 函数名、类命、模块名和包名必须遵守基本命名规则，违背它会导致程序错误；其他常用命名习惯不是必须遵守的，它只是一种编程技巧，遵守它会使得程序更加清晰、简明。

2.1.3　变量的创建

变量在使用前需要创建。在 Python 中，创建变量的语法格式为：

变量=赋值

由于 Python 是动态语言，变量在创建时不需要预先定义其数据类型，但需要对其赋初值。赋值操作通过等号(赋值符号)实现，其目的是将等号右侧的值和等号左侧的变量名进行关联。如果左侧的变量不存在，则将创建该变量并对其赋初值；否则，将用等号右侧的值更新等号左侧变量的原值。

然而，赋值操作并不会改变右侧的任何变量值，它只是将右侧的值与左侧的变量名建立新的关联关系。例如，A＝B＋2，并不会改变 B 的值，而只会将 B＋2 的结果重新赋值给 A。实际上，赋值语句的计算过程可分为两部分：

(1) 首先，计算等号右侧的表达式。

(2) 然后，将计算得到的右侧表达式的值与左侧的变量名相关联。

【例 2-3】 判断下面各赋值语句是否正确。其中，x＝1。

(1) y＝(x＝x＋1)

(2) 5＝x＋1

(3) y＋2＝10

(4) print(y＝5)

解答：

(1) 错误。因为赋值语句是没有返回值的，所以不能将赋值语句再赋值给变量。

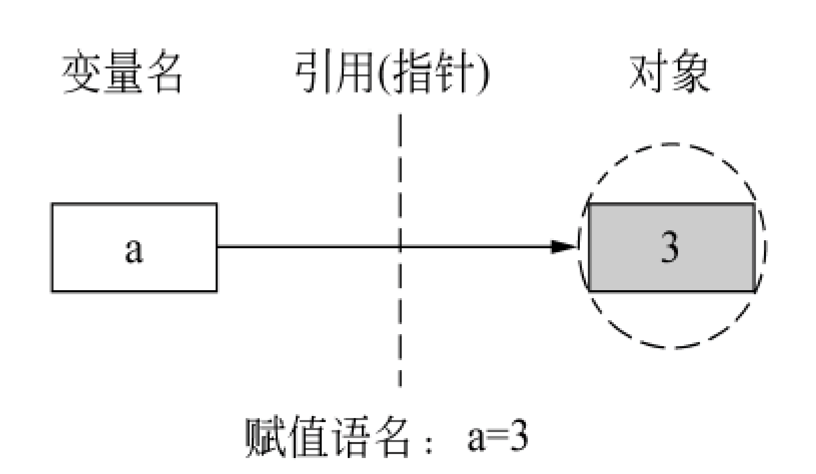
(2) 错误。赋值语句左侧必须是一个合法的变量名(详见 2.1.2 节)，而 5 是常量。

(3) 错误。y＋2 是一个表达式，不是一个合法的变量名。

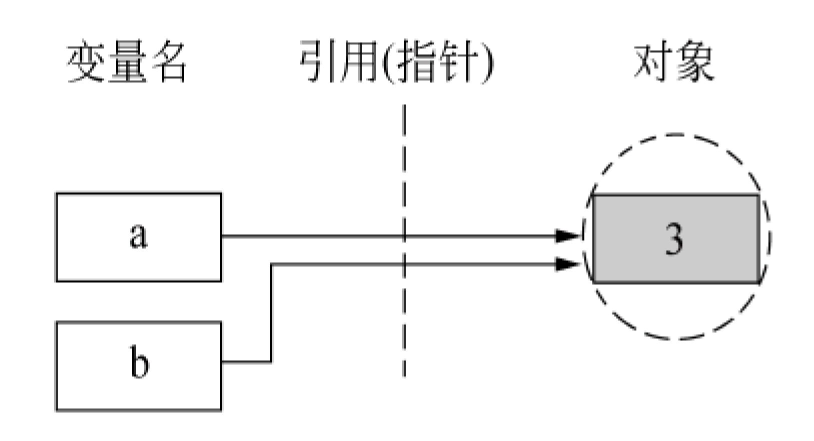
(4) 错误。print 输出语句是在有值后，才会将其输出。但 y＝5 是赋值语句，它不能返回值。

由此可知，赋值语句不能被应用在希望得到值的地方，因为赋值语句本身不返回值。

注意：程序中的赋值语句和数学中的等式是完全不同的概念。赋值操作并不是将右侧的值直接赋值给左侧的变量，而是建立两者之间的关联关系。实际上，赋值操作是将右侧的值的引用赋值给了变量，如图 2-3 所示。因此，赋值语句是没有返回值的。实际上，变量是指向某个对象所在内存空间的一个指针。



(a) a=3，运行后变量a变成了对象3的一个引用即指针



(b) a=3，b=a，运行后，变量a和变量b指向同一个对象的内存空间

图 2-3　变量与赋值



说明：在 Python 中有以下 4 种赋值方式。

(1) 赋值操作符赋值：A＝3，将右侧的值和左侧的变量名关联起来。

(2) 增量赋值：x＋＝1 等价于 x＝x＋1，将右侧算式计算的结果赋值给左侧的变量。

(3) 多重赋值：x＝y＝z＝1，将最右侧的值分别赋值给多个变量。

(4) 多元赋值：将多个变量同时赋值给多个变量，如两个元组间的赋值。

【例 2-4】 读取变量的值。

(1) 执行下面语句后，a＝？ b＝？

```
a=3
b=a
a=4
```

(2) 执行下面语句后，a＝？ b＝？

```
a=1
b=1
b=a+b
a=a+b
```

解答：(1) 首先，变量 a 被赋值为 3。然后，变量 b 被赋值为变量 a，即变量 b 和变量 a 指向同一个内存空间，此时变量 b 的值为 3；最后，变量 a 又被赋值为 4，即变量 a 指向了一个新的内存空间，此时变量 a 的值为 4，但变量 b 依然指向原来的内存空间，其值不变仍为 3。因此，答案为：a＝4，b＝3。

(2) 变量被赋值的原理和(1)相同，最终执行后的答案为：a＝3，b＝2。

2.2　数据类型

变量的作用是将现实世界的各类事物输入计算机中，为利用计算机解决实际问题提供必要的数据支撑。在创建变量的过程中，要考虑变量的数据类型。数据类型是编程的基础。程序设计的本质就是对数据进行处理。

根据自然属性,现实世界的事物可以被分为数字、逻辑、字符串、图形、图像、音频和视频等多种类型。被计算机处理的数据来源于现实世界,这些数据也必然以分类的形式存在。但是,现实世界和计算机世界既相互联系,又相互区别。在计算机中,变量仅有数值类型和非数值类型两种类型。因此,现实世界的事物不能被直接映射到计算机世界的变量中,需要对其进行抽象、分类和转换。其中,数字、逻辑被抽象为数值类型;字符串、图形、图像、音频和视频等则被抽象为非数值类型。

说明:数据在计算机中主要有数值型变量和非数值型变量两大类表示形式。数值型变量是指被人为定义的数字(如整数、小数、有理数等)在计算机中的表示,这种被定义的数据形式可直接载入内存或寄存器,进行加、减、乘、除的运算,一般不经过数据类型的转换,所以运算速度快,具有计算意义。另一种非数值型的数据,例如字符型数据(如'A'、'B'、'C'等),是不可直接运算的字符在计算机中的存在形式,具有信息存储的意义。

计算机只能识别二进制数字,因此,所有非数值型数据在输入时都会被转换成一种特殊的数值型数据,即 ASCII 码。每个 ASCII 码对应一个非数值型的数据。

2.2.1 数值类型

数值类型是指能够被直接存储到内存或寄存器中的数据。数据从现实世界被输入到计算机世界时,其数字的本质特征没有发生改变,只是在形式上从十进制转换成了二进制。

根据表达含义,又可将数值类型分为数字类型和布尔类型。

1. 数字类型

Python 3.x 提供的数字类型有整型(int)、浮点型(float)和复数(complex),如表 2-2 所示。

表 2-2 Python 3.x 中数字类型

类型	描　　述	实　　例
整型	表示正整数或负整数	234,-128,0b1(二进制),0o7(八进制),0x15(十六进制)



续表

类型	描　述	实　例
浮点型	表示实数	2.7,123e+10,8e-3,-23e+5
复数	表示由实部和虚部构成的数	1+2j,3.1j,3e+26j

注：表中的字母不区分大小写,如 0b1 也可写成 0B1。

说明：(1) 使用 Python 时,你会惊讶地发现,它可以表示无限长度的数字,即 Python 具有大数计算功能,可以直接计算出“27392361983108271361039746313 * 37261038163103818366341087632113=”这样大数相乘的结果,而不会发生由于数据位数超过所能表达的范围而产生的溢出的情况。其原因在于：Python 3.x 具有无限精度的整型,即 Python 3.x 不再区分整型和长整型(表示无限大的整数)。如果输入的数值非常大,已经超过了整型所能表示的范围,Python 3.x 会自动将其转换为系统可识别的长整型。该长整型数据所能处理的范围依赖于(虚拟)内存大小。

(2) 计算机的内存容量是有限的,因此,它所能表示的数字范围也是有限的。普通整型的取值范围和机器字长(位数)直接相关。例如,在 32 位机器上,其取值范围是 $-2^{32-1}\sim2^{32-1}-1$,即在[-2147483648,2147483647]区间内的整数;在 64 位机器上,其取值范围为 $-2^{64-1}\sim2^{64-1}-1$,即在[-9223372036854775808,9223372036854775807]区间内的整数。但是,这种限制并不适用于 Python 中的数据表示,Python 具有大数处理功能。

【例 2-5】 显示学生成绩单。

学生成绩单中应显示各科成绩、总分和平均分等内容。

```
Num=3
Math=95.4
English=86
Computer=73.6
Sum=Math+English+Computer
Average=Sum / Num
print("The sum score is",Sum)
```

```
print("The average score is", Average)
```

学生各科成绩可能会出现小数的情况,因此用浮点类型表示;而科目个数则是一个整型变量。Python 支持混合模式的算术运算。当使用浮点类型与整型运算时,结果为浮点类型,如 2+1.0=3.0;当使用浮点类型与复数类型运算时,结果为复数类型,如(3+0j)+4=7+0j。如果使用不兼容的数据类型进行运算,那么会抛出 TypeError 异常。

计算时,浮点类型需要注意的主要问题就是误差。例如,除不尽时会对无限小数进行截断,2/3=0.6666666666666666。另外,在 Python 中执行 0.2+0.1,会得到 0.30000000000000004 的奇怪结果。Python 中浮点数计算产生误差的原因有两点:①是计算机的内存空间有限,无法存储长度无限的数。②任何类型的数据在计算机中都是采用二进制形式存储的。Python 中的浮点数是采用二进制分数表示的,而程序中的浮点数是采用十进制分数表示的,这就导致十进制分数在转换为二进制分数时存在尾数上的截断误差,即 0.1 在计算机中存储的是其近似值。

2. 布尔类型

人们可能听过流传的一句话:"人都是要死的,苏格拉底是人,所以苏格拉底是要死的",这里面蕴含着深刻的逻辑思维。逻辑是人的一种抽象思维,是运用思维法则和规律来理解和区分客观世界的思考问题的过程。但思维是存在于人脑中的无形之物,且每个人的思维都各不相同。那么,我们能否用一种普遍、恰当的符号系统来描述所有问题并模拟整个思考过程呢?英国人乔治·布尔把逻辑学和数学相结合,提出思维的可计算思想,即利用数学中的计算来模拟人的逻辑思考过程,将其符号化、形式化和计算化。

逻辑是一个真与假的世界,它仅有两个值:逻辑"真"和逻辑"假"。在数学中,分别用非 0(通常是 1)和 0 表示;在计算机中,分别用 True(或一个非 0 值,通常是 1)和 False(或 0)表示,即被称为布尔类型。编程时,逻辑命题被翻译成由布尔类型的变量(简称逻辑变量)和逻辑运算符(详见 3.1.2 节)构成的逻辑表达式(详见第 4 章),通过逻辑运算得到的逻辑值来判断事件结果,最终实现逻辑推理的计算机化。

注意:在 Python 中,布尔类型的变量只能被赋值为 True(通常为 1)或 False(或 0),

不能写成 true 或 false。

【例 2-6】 判断一个整数是否是偶数。

解答：在整数中，能被 2 整除的数是偶数，否则是奇数。因此，当使用计算机判断整数的奇、偶性时，需要将题目描述为一个逻辑表达式，该逻辑表达式常常是由“如果……那么……”构成的因果条件关系。本题的伪代码形式如下：

```
A=输入的整数
如果"A 模 2==0",则:
    证明该正整数是一个偶数
```

运行时，计算机计算出“A 模 2”的结果，并将其与 0 进行比较。如果该余数等于 0，即“如果”关键字后的条件语句的值为真，那么“则”关键字后的结果语句会被执行；否则，“则”关键字后的结果语句不会被执行。

【例 2-7】 计算圆面积和周长，要求各计算所需参数由用户自行输入。

解答：这是一个很简单的程序设计题目，只需要弄清楚输入数据的类型、采用何种计算方法，最后直接将结果输出即可。程序代码如下所示：

```
r=input('please input your radius:')
r=int(r)
c=2*3.14*4
a=3.14*(r**2)
print('circumference',c)
print('area',a)
```

说明：Python 3 提供 input([prompt])输入函数，该函数接收一个标准输入数据，返回为 string 类型，即字符串类型(string 类型)，prompt 为提示信息，是字符串类型。按下 Ctrl+Z 结束输入。例如，strText=input("Input a string:\n")：接收输入数据作为字符串类型传给 strText，\n 为提示信息换行。

思考：上述程序完全正确吗？是否存在缺陷呢？比如当半径被误输入为负数时，结果还正确吗？在解决通过用户输入的方式来获取数据的交互式问题时，一定要注意程序

的鲁棒性,即程序应该有对于规范要求以外的异常输入情况的处理能力。因此,我们要对用户输入的数据进行合理的判断和限定,修改后的程序如下所示:

```
r=input("please input your radius: ")
r=int(r)
if(r>0):
    c=2*3.14*4
    a=3.14*(r**2)
    print('circumference',c)
    print('area',a)
else:
    print("the radius must be positive")
```

由此可见,写对一个程序绝非易事。没有最好的程序,只有更好的程序!编程时,要兼顾问题的抽象程度、编程者的能力水平和程序的执行效率等多方因素,既不能将问题无限地复杂化,也不能完全忽略异常情况,要适度、适量和适当!笼统地说,一个正确的程序就是能准确无误地完成它被期望赋予的功能。一个小程序,其实能反映一个人的思维方式和水平。

2.2.2 非数值类型

1. 字符串类型

现实世界不仅需要用数字精确地传达信息,还需要用文字描述事物、交流情感等。在计算机中,表示和存储文本的数据类型称为字符串类型。

从形式上,字符串类型是一个由 0 个、1 个或多个字符组成的集合类型。集合类型是指包含多个对象并将其组织起来成为一个对象的类型。从含义上,字符串类型可以理解为是一种字符序列,它将字符集合按照一定顺序组织在一个序列中。从内容上,字符串可以包含数字、字母和常用控制字符(如转义符,如表 2-3 所示)和汉字。



表 2-3　常见转义字符

转义字符	含　　义	转义字符	含　　义
\（在行尾时）	续行符	\\	表示反斜杠符号\
\n	换行符	\'	表示一个单引号，不是字符串结束标识
\r	回车	\"	表示一个双引号，不是字符串结束标识
\t	横向制表符	\a	响铃
\v	纵向制表符	\f	换页
\e	转义	\b	退格（Backspace）

Python 采用“所输即所得”的方式来定义字符串变量，主要有三种表示字符串的方法。

1）单引号表示法

将字符串内容放在一对单引号中间，如 a='hello'。

2）双引号表示法

将字符串内容放在一对双引号中间，如 b="HelloPython"。

在 Python 中，单引号表示法和双引号表示法在字符串显示上完全相同，一般不用区分。但是通常情况下，单引号用于表示一个单词，双引号用于表示一个词组或句子。

3）三引号表示法

将字符串内容放在一对三引号中间。三引号不仅保留字符串的内容，还保留字符串的格式。三引号通常用来输入多行文本信息，一般可以表示大段的叙述性字符串。例如，电视剧唐顿庄园的主题曲 *Did I Make The Most of Loving You*，定义字符串变量 Song：

```
Song="""So many things we didn't do.
Did I give you all my heart could give?
Two unlived lives with lives to live.
When these endless, lonely days are through,
I'll make the most of loving you….  """
```

注意：单引号、双引号、三引号的异同。

(1) 三种引号都必须成对出现，并且可以相互嵌套，但都不能嵌套自己。判断字符串定义是否正确的简单方法，就是在 Python 提供的 IDLE 环境中进行测试，如果系统没有报错，则定义方法正确。

(2) 单引号或双引号所表示的字符串默认都是写成一行，如果想写成多行，则需要使用\(连行符)，其效果和直接使用三引号相同。

转义字符，顾名思义，就是将反斜杠“\”后面的字符转换成另外的意义。例如“\n”，“n”不代表字母 n 而作为“换行符”；而如果想在屏幕上输出一个单引号，则必须用“\'”，即将字符串的结束标识转义为普通的单引号。

另外，在计算机中，字符串是以字符为单位按照从左到右的顺序依次进行存储的，如图 2-4 所示。字符串的名字指向存储字符串首字符的内存单元。其中，每个字符都有自己固定的位置，占据一个字节空间，称为字符串中的一个元素。在 Python 中，字符串可以通过下标和方括号“[]”的组合来访问指定位置上的元素，且下标是从 0 开始计数的，如 s="buaa"，s[1]="u"。另外，Python 也可以使用负数下标访问字符串中的元素，如[-1]表示最后一个元素，[-2]表示倒数第二个元素，等等。

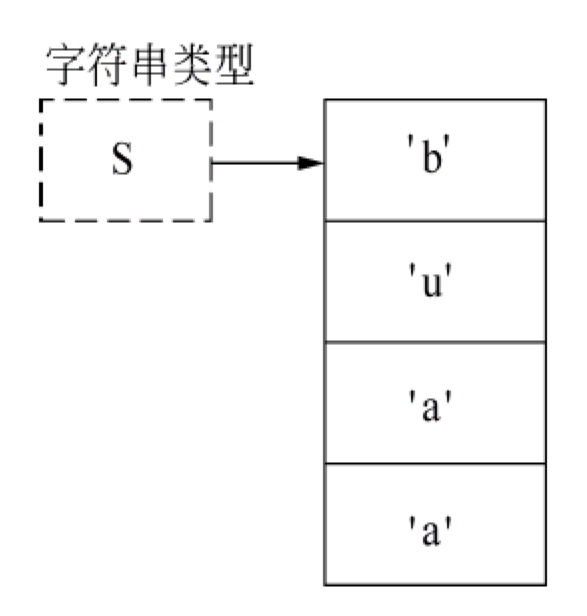


图 2-4　字符串在内存中的存储

这里需要注意，在 Python 中，字符串一旦声明就不能再改变了，即 Python 中的字符串实际上是一个具有固定长度的字符序列。因此，它不需要结束标志，并且对字符串的修改其实是生成了一个新字符串。

【例 2-8】 制作通讯录。

通讯录包括联系人的姓名、电话、工作单位、通信地址等信息。

```
Name='Heather'
Tel='13000000000'
Affinity="School of Computer Science and Engineering, Beihang University"
```



```
Address="""Beihang University
37 Xueyuan Rd. Haidian District Beijing 100191, P. R. China"""

print('Name:',Name)
print('Telephone:',Tel)
print('Affinity:',Affinity)
print('Address:',Address)
```

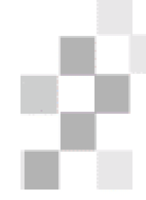
2. 多媒体

所谓多媒体，就是多种媒体的融合，它将文字、声音、图形、图像、视频等多种媒体集成到计算机中，使信息具有声、图、文并茂的表现形式。但是，计算机是采用二进制存储和处理信息的，因此，各种类型的信息在输入计算机时都必须进行数字化处理，即将它们转换成由一组 0 和 1 组成的二进制编码。

说明：不同类型的信息有不同的编码格式，但是它们都要经过数字化才能被计算机存储、传输和处理。信息的数字化过程就是把自然界连续的模拟量变成离散的数字量。其过程是先将连续的模拟量切分成一个个离散的点，然后用二进制表示这些点的值。最后，将这些离散点所对应的二进制编码依次存储在一个文件中，这就形成了数字化的声音和图像。

这些在计算机中以二进制形式存储的数据称为对象。在 Python 中，事事皆对象。要想将多媒体信息输入到计算机中，必须为它们创建对象。对象其实就是为存储各类数据而被分配的一块内存空间，变量就是通过赋值方式建立的对对象的引用。文本和数字的对象创建较为简单，直接通过“所输即所得”的方式进行赋值；而声音、图形、图像和视频的对象创建相对比较复杂，需要借助内置模块或第三方库文件所提供的多媒体处理函数实现，其语法格式如下：

```
import 多媒体处理模块 #导入 Python 内置模块或第三方库文件
多媒体对象=内置模块.方法() #生成对象
多媒体对象.方法() #调用内置模块提供的方法实现对多媒体对象的处理
```



说明：对象和变量可以如下理解。将一个值赋值给一个变量的过程可以理解为将一个值放到一个盒子中，然后给盒子贴上一个标签；将一个变量赋值给另一个变量的过程可以理解为在原来的盒子上又贴了一个新的标签；将一个变量赋一个新值的过程可以理解为将新值放到一个新的盒子中，并撕下原盒子的标签将其贴到新盒子上。这里，装值的盒子就是对象，而盒子上的标签就是变量。对象是一个真实存在的、具有实际意义的事物，而变量仅是一个为方便数据操作而创建的名字。

【例 2-9】 绘制一个圆形。

```
import turtle #导入内置图形处理模块，turtle
myCircle=turtle.Turtle() #生成一个对象
myCircle.circle(100, -360) #绘制特定的圆形
```

本例中，myCircle 是一个由内置模块 turtle 生成的图形对象。程序通过 myCircle 对象调用 turtle 内置模块提供的 Turtle()方法，完成指定圆形的绘制。

计算机是没有思维的，如果想让它绘制一个图形，那么就必须告诉它所绘图形的特征和绘制图形的步骤等信息。在本例中，我们直接使用 Python 3. x 的内置图形处理模块 turtle 完成圆形的绘制。在执行过程中可以看到，很多绘制细节(速度、方向、位置等)都是计算机自动完成的，提高了编程效率，这就是内置模块的作用。内置模块提供了很多实用方法，可供编程人员直接使用。

【例 2-10】 转换图片格式，将图片从 jpg 格式转换为 png 格式。

```
from PIL import Image #导入外部图像处理库
myImage=Image.open("C:\Users\new\Desktop\view.jpg") #生成图像对象
myImage.save('convert.png') #将图片转换为 png 格式并存储在程序所在路径
myImage.show() #显示图像对象
myImage.close() #销毁图像对象
```

PIL(Python Imaging Library)是 Python 提供的一个强大、方便的图像处理库，但是它仅支持到 Python 2.7。因此，在 Python 3. x 中，我们需要安装第三方库文件(通常是 pillow 库文件)来实现对图像的各种操作。在本例中，myImage 是一个由库 PIL 中的



Image 类生成的图像对象。程序通过 myImage 对象调用 Image 类提供的 open()方法，生成一个指定图像的对象。然后，利用 Image 类提供的各种现成方法实现相应功能。最后，采用 close()方法销毁图像对象。

提示：(1) Python 3.x 在图像处理时，通常使用 pillow 库。但 pillow 是一个第三方库，它在使用前，需要手动安装，其下载地址为 https://pypi.python.org/pypi/Pillow/3.0.0，可以根据本机上的 Python 版本，选择合适的 pillow 安装包。

(2) 在使用 pillow 库的 Image 内置函数时，需要添加 from PIL import Image 语句。

【例 2-11】 播放音频文件。

```
import winsound #导入音频处理内置模块
soundFile="C:\Users\new\Desktop\we are the world.wav" #定义音频对象
winsound.PlaySound(soundFile, winsound.SND_FILENAME) #播放指定的音频文件
```

在本例中，soundFile 是一个字符串。程序通过调用 winsound 内置模块提供的 PlaySound()方法，播放指定的声音文件。其中，PlaySound()方法需要设置相关参数。

说明：(1) winsound 是 Python 3.x 提供的内置模块，它通过调用系统的 API 播放音乐文件。它目前只能播放 *.wav 格式的音频文件，不能播放 *.mp3 格式的音频文件。

(2) PlaySound(sound, flags)是 winsound 模块提供的音频播放函数。其中，第一个参数 sound 是 *.wav 音频文件的完整路径；第二个参数 flags 是对 *.wav 文件做何种操作的解释，它有不同的取值，当将 flages 设置为 SND_FILENAME 时，其作用是指明第一个参数的含义是一个 *.wav 文件名。

【例 2-12】 播放视频文件。

```
import cv2 #导入外部视频处理库
myVideo=cv2.VideoCapture("E:/Megamind.avi")#定义视频对象
if myVideo.isOpened(): #判断视频对象是否能被打开
    while True: #循环播放
        ret, prevframe=myVideo.read() #视频解码并返回当前帧
```

```
        if ret==True: #如果当前帧被成功读取
            cv2.imshow('video', prevframe)#将当前帧显示在窗口上
        else: #否则
            break #跳出循环
cv2.destroyAllWindows() #销毁视频播放窗口
```

Python 目前还没有能处理视频的模块。通常,它需要借助 OpenCV 提供的接口操作视频文件。在本例中,myVideo 是一个由第三方库文件 cv2 生成的视频对象。程序通过调用 cv2 模块提供的 VideoCapture()方法,打开指定的视频文件。视频是由若干帧组成的,按序播放各帧就能实现视频的播放。设计时,采用循环方式,通过 read()方法依次获取视频对象中的每一帧,并做如下判断:如果当前帧读取正确,则将其显示在窗口中;否则,跳出程序。最后,采用 destroyAllWindows()方法销毁视频播放窗口。

提示: cv2 库中的 read()方法能够对视频文件进行解码并同时返回两个值,即一个结构。其中,第一个值是一个用于判断当前帧是否被成功读取的标志,它是一个布尔类型数据;第二个值用于获取当前帧,它是一张图片。

习　　题

1. 下列哪些是 Python 中正确的变量名?

(1) 1_myname

(2) Python~3.x

(3) class

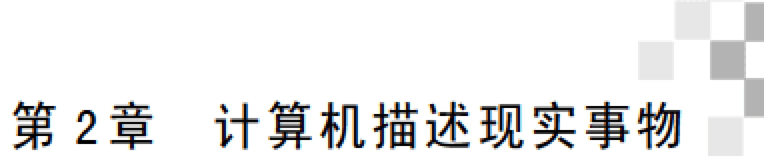
(4) ——a

(5) stuId

(6) y—2

(7) my name

2. 判断下面各赋值语句是否正确,并阐述错误原因。其中,a=2。



(1) a＝b＝c＝d＝1

(2) b＝a＝a＋1

(3) b＝(a＝1＋3)

(4) 0＝a * 0

(5) a * 1＝a

(6) print (a＝a)

3. 阅读程序并指出程序的输出值。

(1) a＝?,b＝?,c＝?

```
a=5
b=a
a=a+1
b=b+3
c=a+b
```

(2) a＝?,b＝?

```
a=3
b=a
print(a+1)
print(b+3)
```

4. 识别输出结果的数据类型。

(1) 3＋1.0

(2) 3＝＝2

(3) '18688806688'

(4) 'h' * 3

(5) 0/2.0

(6) 1＋0.2－0.2

(7) 1＋0j

5. 输出如图 2-5 中显示的内容。

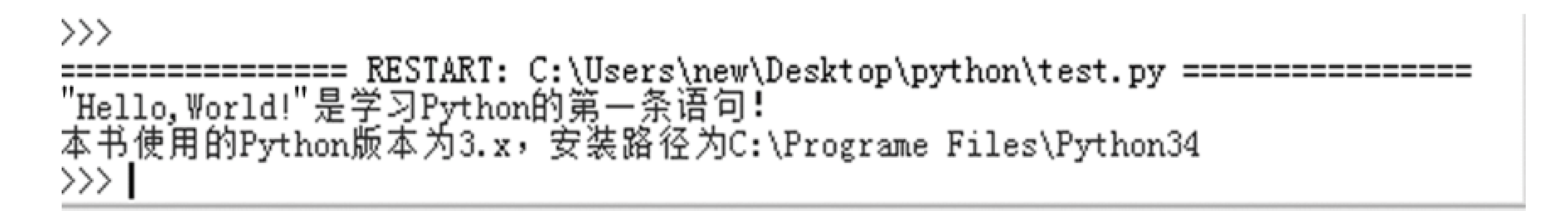


图 2-5　文本输出练习

6. 制作个人简历。个人简历主要包括姓名、性别、年龄、电话、邮箱、个人特点、个人照片等内容(建议使用转义字符增强内容的可读性)。

7. 输出一张图片。

8. 播放一首音乐。

9. 理解 Python 中变量名的含义。

10. 理解 Python 中对象的应用。

第3章　计算机处理现实事物



计算机的作用就是通过对各类数据的处理(计算)来解决实际问题。在Python中,数据是分门别类、按类计算的,即对于不同类型的数据,采用不同的运算方法,具有不同的操作含义。各种复杂问题的处理,都是通过将其分解成计算机所能理解的简单操作实现的。因此,对问题的抽象和分解在实际编程中有着十分重要的作用。计算机根据它所能识别的数据类型,将操作分为数值类型操作和非数值类型操作。数值类型操作又可细分为数字操作和布尔操作;非数值类型的操作又可细分为与文本处理相关的字符串操作和与多媒体信息处理相关的图像操作、音频操作和视频操作等。

3.1　数值类型操作

3.1.1　数字操作

数字操作主要用于数值类型的变量,所有在数学中定义的算术运算,例如加、减、乘、除、取余等,都可以通过Python提供的数字操作实现,Python常用的算术运算符如表3-1所示。Python中的运算符和数学中的运算符类似,也区分优先级。各类运算符的优先级如表3-7所示。

Python中算术运算符的计算规则为:①从左到右依次计算;②优先级高的先计算;③同等优先级的按从左到右的顺序计算;④可以用小括号修改低优先级运算符的级别,

使其具有高优先级。

表 3-1　Python 常用的算术运算符

运　算　符	描　　述	实　　例
**	乘方	x**2
*	乘法	x * y
/	除法,返回浮点数	x/y
%	取余	x%y
//	取整数,返回商的整数部分	x//y
+	加法	x+y
−	减法	x−y

另外,Python 3. x 还提供了很多用于数字操作的内置函数,以方便用户使用,具体如表 3-2 所示。

表 3-2　Python 提供的用于数字操作的内置函数

函　数　名	描　　述	实　　例
abs(x)	取绝对值	abs(−12)=12
int(x)	返回 x 的整数部分	int(−3.8)=−3
round(x)	返回 x 四舍五入后得到的整数	round(−3.8)=−4
float(x)	将 x 转换为浮点数	float(6)=6.0
complex(re, im)	返回一个复数,其中 re 为实部,im 为虚部	complex(1, 2)=(1+2j)
c. conjugate()	返回复数 c 的共轭复数	c=4+3j c. conjugate()=(4−3j)
divmod(x,y)	返回一个数值对(x//y, x%y)	divmod(5, 3)=(1,2)
pow(x,y)	求 x 的 y 次幂	pow(2,4)=16
round(x,n)	控制 x 的精度,返回具有 n 位小数的 x	round(3.1415926, 3)=3.142

【例 3-1】　设计一款贷款计算器,比较等额本金和等额本息两种方式下,每月应还本付息的金额,即月供额分别是多少(从 Python 的 Shell 界面输入和显示信息)。2016 年房



贷基准利率如表 3-3 所示。

解答：对于贷款，银行提供两种贷款方式，分别是“商业贷款”和“公积金贷款”；而对于还款，银行提供两种还款方式，分别是“等额本金”和“等额本息”还款。因此，总共有 4 种贷款和还款的组合方式，分别是：商业贷款且等额本金还款、商业贷款且等额本息还款、公积金贷款且等额本金还款、公积金贷款且等额本息还款。然后，我们根据这 4 种不同的组合方式，可以计算出相应的月供额。另外，我们还可以通过比较了解哪种贷款和存款的组合方式比较划算。

计算月供额的主要公式有：

(1) 等额本金的每月应还本付息金额＝[贷款本金×月利率×(1＋月利率)^还款月数]÷[(1＋月利率)^(还款月数－1)]。

(2) 等额本息的每月应还本付息金额＝(贷款本金÷还款月数)＋(贷款本金－已归还本金累计额)×每月利率。

(3) 另外，月利息＝年利息÷12。

表 3-3　2016 年房贷基准利率

贷 款 期 限	年利率/%
一、商业贷款	
一年以内(含一年)	4.35
一至五年(含五年)	4.75
五年以上	4.90
二、公积金贷款	
五年以下(含五年)	2.75
五年以上	3.25

具体代码如下：

```
loanType=input("Please select your type of loans(商业贷款 or 公积金贷款):\
n")
payType=input("Please select your type of payment(等额本金 or 等额本息):\
```

```
n")
loanAmount=float(input("Please input your amount of loans:\n"))
loanMonth=int(input("Please input the number of loan month:\n"))
payBack=input("Please input your payment of loan:\n")
#商业贷款且等额本金还款
if loanType=="商业贷款" and payType=="等额本金":
    if loanMonth<=12:
        monthPay=(loanAmount*4.35/12*(1+4.35/12)**loanMonth)/
((1+4.35/12)**loanMonth-1)
    if loanMonth >13 and loanMonth <=60:
        monthPay=(loanAmount*4.37/12*(1+4.75/12)**loanMonth)/
((1+4.75/12)**loanMonth-1)
    if loanMonth >60:
        monthPay=(loanAmount*4.90/12*(1+4.90/12)**loanMonth)/
((1+4.90/12)**loanMonth-1)
#商业贷款且等额本息还款
if loanType=="商业贷款" and payType=="等额本息":
    if loanMonth<=12:
        monthPay=loanAmount/loanMonth+(loanAmount-payBack)*
4.35/12
    if loanMonth>13 and loanMonth<=60:
        monthPay=loanAmount/loanMonth+(loanAmount-payBack)*
4.75/12
    if loanMonth>60:
        monthPay=loanAmount/loanMonth+(loanAmount-payBack)*
4.90/12
#公积金贷款且等额本金还款
if loanType=="公积金贷款" and payType=="等额本金":
    if loanMonth <=60:
        monthPay=(loanAmount*2.75/12*(1+2.75/12)**loanMonth)/
((1+2.75/12)**loanMonth-1)
    if loanMonth>60:
```



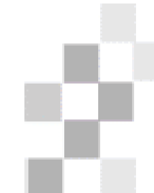
```
            monthPay= (loanAmount * 3.25/12 * (1+3.25/12)**loanMonth)/
((1+3.25/12)**loanMonth-1)
#公积金贷款且等额本息还款
if loanType=="公积金贷款" and payType=="等额本息":
    if loanMonth <=60:
            monthPay=loanAmount/loanMonth+ (loanAmount-payBack) *
2.75/12
    if loanMonth >60:
            monthPay=loanAmount/loanMonth+ (loanAmount-payBack) *
3.25/12
print("the payment of month is",monthPay)
```

计算机常用二进制形式表示数值，而数学中常用十进制表示数值。因此，Python 提供了用于二进制数字操作的位运算符，如表 3-4 所示。

表 3-4　Python 提供的用于二进制数字操作的位运算符

运算符	描　述	实　例
~	按位取反，即～x=－(x+1)	～5=－6
<<	按位左移	5<<2=20，将 101 向左移动 2 位，得 10100
>>	按位右移	5>>2=1，将 101 向右移动 2 位并去掉小数部分，得 1
&	按位与	5 & 3=1，将对应的二进制数执行按位与操作，即 101 & 011=001=1
|	按位或	5 | 3=7，将对应的二进制数执行按位或操作，即 101 | 011 =111=7
^	按位异或	5 ^ 3=6，将对应的二进制数执行按位异或操作(对位相加，不进位)，即 101 ^ 011=110=6

说明：按位运算是程序设计中对二进制数的一种操作。按位运算符的操作数可以是二进制形式，也可以是十进制形式。如果操作数是十进制表示方式，则计算时，按位运算符要先将十进制数转换为二进制数，然后进行相应的按位运算，最后再将计算结果转换为十进制数输出。在计算机系统中，二进制数值一律用补码来表示(存储)。



3.1.2 布尔操作

布尔类型是一种特殊的数值类型，它仅有 True（可用非 0 表示）和 False（可用 0 表示）两个值。由于布尔类型可以使用数值表示，表面上，它可以参与各种数字操作。例如，False＋1＝1，True－2＝－1。但实质上，布尔类型和普通数值类型所表达的含义不同，它不是表示数量，而是表示真或假的逻辑关系，经常用在条件判断中（详见 4.1 节）。布尔类型有自己的特殊操作，例如关系运算和逻辑运算等。其中，关系运算的返回值是布尔类型，如表 3-5 所示；而逻辑运算中操作数（或表达式）的值和返回值均是布尔类型，如表 3-6 所示。这里，需要注意双等号“＝＝”表示判断两个变量是否相等，而单等号“＝”是赋值运算符。

表 3-5　关系运算符（假设：x＝1，y＝2，z＝1）

运算符	描　　述	实　　例
＜	判断左操作数的值是否小于右操作数的值，如果是则条件为真	x＜y，结果为 True
＜＝	判断左操作数的值是否小于或等于右操作数的值，如果是则条件为真	x＜＝y，结果为 True
＞	判断左操作数的值是否大于右操作数的值，如果是则条件为真	x＞y，结果为 False
＞＝	判断左操作数的值是否大于或等于右操作数的值，如果是则条件为真	x＞＝y，结果为 False
＝＝	判断左操作数的值是否等于右操作数的值，如果是则条件为真	x＝＝z，结果为 True
!＝	判断左操作数的值是否不等于右操作数的值，如果是则条件为真	x!＝y，结果为 True
is	判断两个标识符是否指向同一个对象	x is z，结果为 False
is not	判断两个标识符是否指向不同的对象	x is not y，结果为 True

说明：(1) Python 中的对象包含三个基本要素：标识符、数据类型和值。其中，对象的标识符就是对象的内存地址，可通过内置函数 id()获得；数据类型是对象的存储类型，可通过内置函数 type()获得；值是对象在内存单元中存储的具体内容，可通过对象本身



（变量名）获得。

（2）Python 中 is 用来判断两个对象的同一性而非相同性。所谓同一性，是指两个对象的 id 相同，即它们指向相同的内存空间。例如，x=1，y=1，x is y 的结果为 False；而 x=1，y=x，x is y 的结果为 True。但是为了节省空间，Python 一般会将值相同的两个变量指向同一个内存地址，使得 x=1，y=1，x is y 的结果为 True。

表 3-6　逻辑运算符

运算符	描　述	实例 （假设：x=True，y=False）
not	逻辑非运算符，反转操作数的逻辑值	not x 结果为 False
and	逻辑与运算符，如果两个操作数都为真，则条件为真	x and y 结果为 False
or	逻辑或运算符，如果至少有一个操作数为真，则条件为真	x or y 结果为 True

布尔操作看似抽象，却在人们的日常生活中应用十分广泛，几乎所有涉及信息检索的地方都会用到布尔操作，例如网上订票、网上购物、网上订餐、路线查询等，可以说互联网中处处都是布尔操作。

【例 3-2】 火车票儿童票购买标准：一名成年人旅客可以免费携带一名身高不足 1.2m 的儿童。如果身高不足 1.2m 的儿童超过一名时，那么只有一名儿童免费，其他儿童必须购买儿童票；儿童身高为 1.2～1.5m 的，必须购买儿童票；身高超过 1.5m 的儿童，必须购买全价座票。

解答：布尔操作与逻辑判断息息相关，解决此类问题的关键就是找到决定条件判断的“条件变量”。“条件变量”一般不止一个。本题中，儿童身高和儿童个数就是两个“条件变量”，它们直接影响着购票类型和购票总价。在应用布尔操作解决问题时，实际问题一般都能被形式化为“如果……则……否则……”句式。本题的伪代码形式如下：

```
priceChild=儿童票票价
priceAdult=成人票票价
shortNum=身高不足 1.2m 的儿童人数
```



midNum=身高在 1.2~1.5m 的儿童人数
tallNum=身高超过 1.5m 的儿童人数
如果 shortNum==1,则:
总票价=priceChild * midNum+priceAdult * tallNum
否则:
总票价=priceChild * (shortNum-1)+priceChild * midNum+priceAdult * tallNum

思考:如何计算"身高不足 1.2m 的儿童人数"?

【例 3-3】 12306 网站预订火车票:购买 2016 年 9 月 1 日,从上海到北京的火车票,如图 3-1 所示。



图 3-1　网购火车票

解答:涉及条件筛选、关键字检索等内容的实际问题,可以通过同时使用比较运算符和逻辑运算来解决。本题的伪代码形式如下:

如果行程=='单程'并且出发地=='上海'并且目的地=='北京'并且出发时间==
'2016-09-01',则:
输出"条件筛选结果"

在本章开始介绍过运算符是有优先级的。当多个运算符同时执行时,必须按照一定的顺序和规则先后执行,才能保证运算的合理性和结果的正确性、唯一性。为了让大家对各运算符间的优先级有一个全面的、整体的认识,我们总结了常用的不同类型运算符之间以及同一类型、不同操作运算符之间的全局优先级列表,如表 3-7 所示。程序运行时,优先级高的运算符先被计算,优先级相同的运算符按照从左到右的顺序依次计算。



表 3-7　常用运算符的全局优先级(从高到低排序)

运　算　符	类　　型	运　算　符	类　　型
**	算术运算符	|	位运算符
~	位运算符	<,<=,>,>=,!=,==	比较运算符
*,/,%	算术运算符	is,is not	比较运算符
+,-	算术运算符	not	逻辑运算符
<<,>>	位运算符	and	逻辑运算符
&	位运算符	or	逻辑运算符
^	位运算符		

3.2　非数值类型操作

计算机的本质就是一台加法器,所有信息都是通过计算进行处理的。这表明不仅数值型数据可以参与计算,非数值型数据也可以在被数值化后进行计算。非数值类型数据的操作包括字符串处理和多媒体处理两大类。

计算机中的计算是区分类型的,不同类型的数据具有不同的计算方法。即使表面上形式相同的计算,也会因其操作对象类型的不同具有截然不同的含义。例如加号"+",对于数值型数据,该计算表示两个操作数相加;而对于字符串型数据,该计算表示两个字符串的连接。这种具有相同计算形式、不同操作功能的方法,在计算机中称为重载。

3.2.1　字符串处理

字符串操作的对象是字符串,即用单引号、双引号或三引号括起来的数据。在现实生活中,经常会用字符串表示信息。例如,系统登录时的用户名和密码、身份验证时的验证码、客房预订时的个人记录等;另外,图片等多媒体信息,在计算机中也常会用二进制字符串表示。这些字符串数据在输入过程中,需要进行字符串比较、内容匹配、文本解析、格式处理等操作。Python 3.x 提供很多对字符串处理的内置函数和操作。

1. 字符串比较

在计算机中,不仅数值型数据可以进行数值间大小的比较,非数值型数据,如字符串,也可以进行字符间“大小”的比较。字符串的比较,就是从两个字符串的下标0开始依次比较对应字符在ASCII表中的顺序(小写字母排序在后、大写字母排序在前,'a'>'A'),顺序在前的字符小于顺序在后的字符。字符串支持的比较操作符为<、<=、=!、=和>=。字符串比较经常被用在字符识别(如车牌号识别)、图像处理(如图像相似度比较)等领域中。

【例3-4】 验证邮箱登录是否成功。

解答:用户登录邮箱时,需要输入用户名和密码,它们都属于字符串。因此,本题旨在判断输入字符串和正确字符串是否相同,伪代码如下:

```
如果邮箱用户名=='注册用户名'并且邮箱密码=='注册密码',则:
    登录成功
否则:
    登录失败
```

具体代码如下:

```
mailname=input("Email username:") #定义用户名
mailpwd=input("Email password:") #定义密码
if mailname=='admin' and mailpwd=='admin2016': #如果用户名和密码都正确,
则登录成功
    print("Login succeed!")
else: #否则,登录失败
    print("Login failed!")
```

2. 字符串匹配

字符串匹配(String Match)问题是计算机科学的基础问题之一,它是指在给定的原字符串和子串(又称模式,Pattern)的输入下,按照一定的匹配条件,输出子串或子串元素在原字符串中出现位置的搜索问题。例如,在原字符串S="abcabcabdabba"中查找子串



T="abcabd",其输出结果为 T 在 S 中首次出现的下标位置,即 3(下标从 0 开始),如图 3-2 所示。字符串匹配问题在实际工程中经常遇到,被广泛应用于各种涉及文字和符号处理的领域中,是网络安全、信息检索、语义分析和计算生物学等重要领域的关键问题。

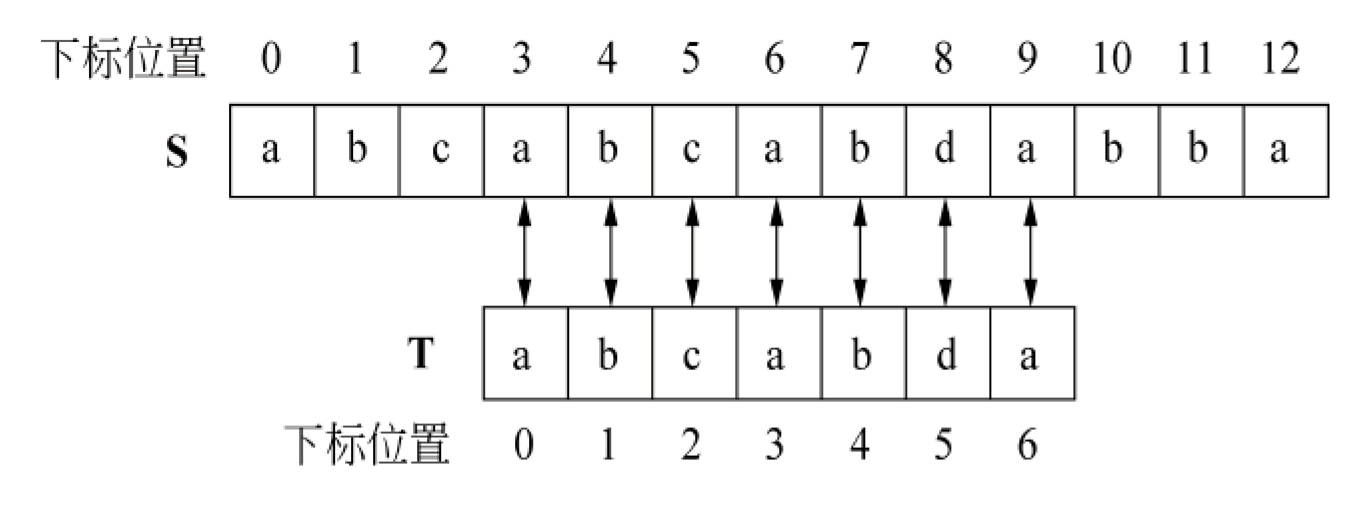


图 3-2　字符串匹配

【例 3-5】 信息检索:在一篇文章中检索指定关键字首次出现的位置。

解答:Python 3.x 提供了内置的字符串匹配函数,可以直接实现在原字符串中检索指定的模式子串。

```
S="在中国人的心中,有一种拼搏叫中国女排,有一种情怀叫中国女排,有一种冠军叫中国女排,这都源于有一种精神叫"女排精神"。"
T="中国女排"
print(S.find(T))
```

说明:Python 3.x 中提供了 4 种内置的字符串匹配函数,分别是 find()、index()、rfind()和 rindex()。其中,find()返回模式子串在原字符串中第一次出现的位置,如果没有匹配项则返回−1;index()和 find()功能相同,但如果没有匹配项并不会返回−1,而是抛出一个异常;rfind()返回模式子串在原字符串中最后一次出现的位置,如果没有匹配项则返回−1;rindex()和 rfind()功能相同,但如果没有匹配项也不会返回−1,而是抛出一个异常。

思考:(1) 如何输出所有匹配的模式子串的位置?

(2) 如果不使用 Python 内置的字符匹配函数,如何实现模式子串的检索?

扩展:Python 提供了更为灵活多样的字符串匹配方法,称为正则表达式。正则表达

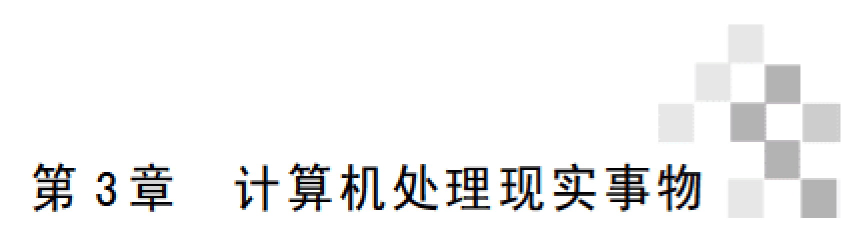
式是一个特殊的字符序列，它能快速检查一个字符串是否与某种模式匹配。使用正则表达式时，需要提前引入 re 模块。re 模块中常用的字符串匹配方法有 re. match、re. search 和 re. sub 等。

3. 字符串操作

字符串作为表示文本的数据类型，在程序中通常以串的整体作为操作对象。字符串操作主要包括字符串连接、替换、删除、截取、复制、比较、查找和分割等，Python 3. x 提供了丰富的用于字符串操作的内置函数，使用这些函数可以大大减轻编程的负担，具体内容如表 3-8 所示。这里需要注意的是，在 Python 中字符串一旦声明就无法修改。因此，对字符串的各种操作都是生成新的字符串，而不是在原字符串上进行修改。例如，a='hello'，a[0]='H'，系统会抛出“TypeError：'str' object does not support item assignment”的错误信息。

表 3-8　Python 提供的用于字符串操作的内置函数

操作类型	字符串操作	描　述	实　例
连接	+	两个字符串连接	'a'+'b'， 输出'ab'
	str. join(String)	用指定子串连接字符串中的各元素	'-'. join('buaa')， 输出'b-u-a-a'
复制	* n	将字符串复制 n 遍	'a' * 3， 输出'aaa'
长度	len(String)	获取字符串长度，即字符串中元素个数	len('Python')， 输出 6
截取	String[index]	截取字符串中下标 index 的元素	'Python'[1]， 输出'y'
	String[index1：index2]	截取字符串中从下标 index1 到下标 index2−1 中的各元素	'Python'[1：3]， 输出'yt'
	String[index1：]	截取字符串中从下标 index1 到结尾的各元素	'Python'[1：]， 输出'ython
	String[：index2]	截取字符串中从开头到下标 index2−1 中的各元素	'Python'[：3]， 输出'Pyt'



续表

操作类型	字符串操作	描　　述	实　　例
去除	String. strip()	去除字符串两端的空格	' buaa '. strip()， 输出'buaa'
	String. lstrip()	去除字符串左端的空格	' buaa '. lstrip()， 输出'buaa '
	String. rstrip()	去除字符串右端的空格	' buaa '. rstrip()， 输出' buaa'
	String. strip(str)	去除字符串两端的指定子串	'HelloPython'. strip('Hello')， 输出'Python'
分割	String. split(str)	按指定字符分割字符串为列表，默认按空格分割	'I like Python'. split()， 输出['I'，'like'，'Python'] '86-010-82111111'. split('-')， 输出['86'，'010'，'82111111']
字符串判断	String. startswith(str)	判断字符串是否以 str 开头	'Python'. startswith('P')， 输出 True
	String. endswith(str)	判断字符串是否以 str 结尾	'Python'. endswith('12')， 输出 False
	String. isalnum()	判断字符串是否全是字母或数字	'Python12'. isalnum()， 输出 True
	String. isalpha()	判断字符串是否全是字母	'Python12'. isalpha()， 输出 False
	String. isdigit()	判断字符串是否全是数字	'Python12'. isdigit()， 输出 False
	String. isupper()	判断字符串中的字母是否全是大写	'BUAA'. isupper()， 输出 True
	String. islower()	判断字符串中的字母是否全是小写	'buaa2016'. islower()， 输出 True
字母处理	String. capitalize()	将字符串的第一个字母大写，其余字母小写	'chiNa'. capitalize()， 输出'China'
	String. title()	将字符串中各单词的首字母大写	'string of python'. title()， 输出' String of Python'
	String. upper()	将字符串全部转换为大写	'buaa'. upper()， 输出'BUAA'

续表

操作类型	字符串操作	描　　述	实　　例
字母处理	String. lower()	将字符串全部转换为小写	'BUAA'. lower()， 输出'buaa'
	String. swapcase()	将字符串中的大、小写字母互换	'bUAA'. swapcase()， 输出'Buaa'
搜索	String. find(str)	返回子串在原字符串中第一次出现的位置，如果没有匹配项则返回−1	'buaa'. find('a')， 输出 2
	String. index (str)	返回子串在原字符串中第一次出现的位置，如果没有匹配项则抛出异常	'buaa'. index('b')， 输出 0
	String. rfind(str)	返回子串在原字符串中最后一次出现的位置，如果没有匹配项则返回−1	'buaa'. rfind('a')， 输出 3
	String. rindex (str)	返回子串在原字符串中最后一次出现的位置，如果没有匹配项则抛出异常	'buaa'. rindex('B')， 抛出一个异常，ValueError：substring not found
	String. count(str)	获得字符串中指定子串的数目	'buaa'. count('a')， 输出 2
替换	String. replace (oldstr，newstr)	将原字符串中 oldstr 子串替换为新的 newstr 子串	'buaa'. replace('a','A')， 输出'buAA'
	String. replace('old','new',maxReplaceTimes)	将原字符串中 oldstr 子串在指定次数内替换为新的 newstr 子串	'buaa'. replace('a','A',1)， 输出'buAa'

【例 3-6】 微信账号用手机号注册时，需要对注册账号做如下判断和处理：注册账号长度为 11 位，必须是数字，不能包含空格，并且以数字'1'开头。

解答：可以通过字符串操作对注册账号进行判断和处理，伪代码形式如下：

```
如果注册账号长度等于 11 并且注册账号全部都是数字并且注册账号以数字'1'开
头，则：
    去除注册账号中的空格并输出
否则：
    输出注册账号格式错误
```



将上述伪代码翻译成具体的 Python 程序，如下表示：

```
account=input("Please input your account:\n")
if len(account)==11 and account.isdigit() and account.startswith('1'):
    print("注册账号输入正确，它是"+account.strip())
else:
    print("注册账号格式错误!")
```

思考：上述程序仅能检测出注册账号的格式是否正确，但是无法显示具体的错误原因，即无法告知用户是输入长度有误还是包含有字母等非法字符，请问如何修改？

【例 3-7】 第 2 章学习了 Python 中变量名的命名规则，请编写程序验证输入的变量名是否正确。

解答：Python 中变量名的命名规则为：①变量名必须以字母或下画线开头；②除首字符外，变量名可以包含任何字母、数字和下画线的组合；③除下画线外，变量名中不能出现其他任何特殊字符，如分隔符、空格、标点符号或运算符号等不能出现；④变量名长度不受限制；⑤变量名区分大小写；⑥变量名不能与 Python 自带的关键字重名。

规则①～③都是用于识别字符串中是否含有非法字符，可以概括为：字符串的首字母必须是以字母或下画线开头；而除首字母外的其他字符必须是由字母、数字、下画线组成。其中，规则①～④都可以通过字符串操作来判断，伪代码形式如下：

```
如果变量名首字符不是字母或者变量名首字符不是下画线，则：
    输出"变量名不符合规则 1"
如果除首字符外的变量名除去下画线后不全是字母或数字，则：
    输出"变量名不符合规则 2"
如果变量名除去下画线后不全是字母或数字，则：
    输出"变量名不符合规则 3"
如果变量名长度大于无穷，则：
    输出"变量名不符合规则 4"
```

将上述伪代码形式翻译成具体的 Python 程序，如下所示：

```
VarName=input("Please input your variable name:\n")
```




```
#判断首字母是否只包含字母或下画线
if not (VarName[0].isalpha() or VarName[0]=='_'):
    print("变量名不符合规则 1")
#判断除首字母外是否只包含字母、数字或下画线
if not (VarName[1:len(VarName)].replace('_','A').isalnum()):
    print("变量名不符合规则 2")
#判断字符串是否只包含字母、数字或下画线
if not (VarName.replace('_','A').isalnum()):
    print("变量名不符合规则 3")
#判断字符串长度是否不受限制
if not (len(VarName)<float("inf")): #float("inf")表示无穷
    print("变量名不符合规则 4")
```

提示：Python 没有提供实现“字符串除去指定字符或子串”的内置函数，String.strip()系列函数都只能在字符串的两端除去指定子串或空格。本题中，我们在判断规则②和规则③时，巧妙地利用了 String.replace(oldstr，newstr)内置函数，将字符串中的下画线强制替换为指定的字母，然后再判断转换后的新字符串是否仅包含字母或数字。

思考：当变量名命名存有错误时，上述程序能逐条检测并且输出错误信息；但当变量名命名正确时，上述程序却无法输出正确结果。请问：如何修改，既能检测出变量名中的所有错误，又能验证出正确的变量名？

4. 字符串转换

Python 提供了很多用于字符串和数字之间转换的内置函数。例如，int()用于将字符串变量转换为整数类型，str()用于将整型变量转换为字符串类型。既然 Python 可以对信息分类存储，为何还要“画蛇添足”，在变量间执行类型转换呢？所谓尺有所短、寸有所长，不同类型的变量在处理问题时虽各有特点，但也不是万能的。例如，数字类型的变量就存在存储长度的限制，不能表示非常大的数；而字符类型的变量在某些情况也需要计算。

另外，Python 中转换的本质并没有把一个变量从一种类型“转换”成另一种类型，而



是创建了一个新的变量。

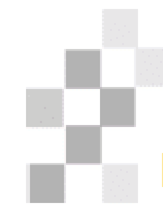
【例 3-8】 简易计算器。通过 Python 的 Shell 界面分别输入操作数和操作类型(+、-、*、/),并且输出计算结果。

解答:方法一:Python 3.x 提供了内置的输入函数 input(),它将数据从界面输入到程序中。但是,input()函数输入的数据一律都是字符串类型,本题需要对输入的数据进行计算,因此必须对其进行数据类型的转换。具体程序如下所示:

```
inData1=input("Please input a number:\n")
data1=int(inData1)
inData2=input("Please input another number:\n")
data2=int(inData2)
opera=input("Please input the operator:\n")
if opera=='+':
    res=data1+data2
    print(data1,opera,data2,'=',res)
if opera=='-':
    res=data1 -data2
    print(data1,opera,data2,'=',res)
if opera=='*':
    res=data1*data2
    print(data1,opera,data2,'=',res)
if opera=='/':
    res=data1 / data2
    print(data1,opera,data2,'=',res)
```

思考:上述程序是否健壮?是否在各种输入包括异常输入等情况下,如除数是 0 或者将除号误写成"\"等情况,都能正常执行或结束?

俗话说:世界上没有一模一样的程序,同一问题可以用不同的方法求解。评价一个程序的好坏,通常以其执行时间(时间复杂度)和所占空间(空间复杂度)等指标来衡量。如何写出高质量的程序?如何将实际问题快速定位到所学知识领域?如何确定解决问



题的方法和思路？这些都需要通过实践不断掌握编程知识，积累编程经验，学会抽象问题、检索知识、调试错误等自学能力。

方法二：针对本题中输入数据的类型均是字符串的特点，思考是否有函数能直接对由字符串构成的表达式进行处理，通过查找，发现内置函数 eval()具有强大的字符串处理能力，它能将字符串当成有效的表达式进行求值并返回计算结果。具体程序如下所示：

```
inData1=input("Please input a number:\n")
inData2=input("Please input another number:\n")
opera=input("Please input the operator:\n")
express=inData1+opera+inData2
res=eval(express)
print(express,'=',res)
```

【例 3-9】 大数相乘。计算：

27392361983108271361039746313 * 37261038163103818366341087632113＝？

解答：Python 具有大数计算功能(详见 2.2.1 节)。因此，可以直接在 Python 的 Shell 界面输入如下表达式“27392361983108271361039746313 * 37261038163103818366341087632113”，得到结果为 1020667845230151490815971954949119846724511883349579392149369。但是，抛开 Python 具有大数计算的特殊能力，针对大数计算问题，我们一般会先将大数转换为字符串，以避免由于数据位数过长而产生的溢出等问题，然后再对字符串类型的被乘数和乘数的每一位分别执行乘法运算，最后将各个位上的乘积求和再拼接成字符串。程序伪代码形式如下所示：

```
strData1=将作为被乘数的大数转换为字符串类型
strData2=将作为乘数的大数转换为字符串类型
将结果存入字符类型的乘积 res 中
遍历字符串类型被除数的每一位,记为 i
    遍历字符串类型乘数的每一位,记为 j
        res [i+j]=strData1[i] * strData1[j]
```



如果 res [i+j] >=10,则产生进位,并对进位进行处理:

res[i+j+1]=乘积的[i+j] /10

res[i+j]=乘积的[i+j] %10

具体代码如下:

```
strData1=str(27392361983108271361039746313)
strData2=str(37261038163103818366341087632113)
res=[0] #由于 Python 中字符串是不可变数据类型,即定义后就不能修改,所以这里将结果定义为列表类型,列表的值是可以修改的
for i in range(len(strData2)):
    for j in range(len(strData1)):
        bitData=int(strData2[len(strData2)-i-1])*int(strData1[len(strData1)-j-1]) #从被乘数和乘数的个位逐个开始相乘
        if i+j>len(res)-1:
            res.append(0) #Python 无法直接定义一个未知长度的列表,只能通过逐次添加 0 元素的方式增长列表
        res[i+j]=res[i+j]+bitData #乘积从个位开始逐渐累加
print(res)
#检查每位数字大于 10 的情况
for k in range(len(res)):
    if res[k]>=10:
        if k+1>len(res)-1:
            res.append(0)
        res[k+1]=res[k+1]+int(res[k]/10) #int(res[k]/10)获得进位并将其添加到高位中
        res[k]=res[k]%10 #获得进位后的余数,留存在低位
res.reverse()
print(res)
```

说明:例 3-9 输出的结果是用列表形式显示的。

注意:(1) 被乘数和乘数按位相乘时,是从个位到高位逐位计算的,但字符串下标 0 的位置是从高位开始标识的,因此在实现数据读取、结果输出等操作时要注意两者之间



如果 res [i+j] >=10,则产生进位,并对进位进行处理:

res[i+j+1]=乘积的 [i+j] /10

res[i+j]=乘积的 [i+j] %10

具体代码如下:

```
strData1=str(27392361983108271361039746313)
strData2=str(37261038163103818366341087632113)
res=[0] #由于 Python 中字符串是不可变数据类型,即定义后就不能修改,所以这里将结果定义为列表类型,列表的值是可以修改的
for i in range(len(strData2)):
    for j in range(len(strData1)):
        bitData=int(strData2[len(strData2)-i-1])*int(strData1[len(strData1)-j-1]) #从被乘数和乘数的个位逐个开始相乘
        if i+j>len(res)-1:
            res.append(0) #Python 无法直接定义一个未知长度的列表,只能通过逐次添加 0 元素的方式增长列表
        res[i+j]=res[i+j]+bitData #乘积从个位开始逐渐累加
print(res)
#检查每位数字大于 10 的情况
for k in range(len(res)):
    if res[k]>=10:
        if k+1>len(res)-1:
            res.append(0)
        res[k+1]=res[k+1]+int(res[k]/10) #int(res[k]/10)获得进位并将其添加到高位中
        res[k]=res[k]%10 #获得进位后的余数,留存在低位
res.reverse()
print(res)
```

说明:例 3-9 输出的结果是用列表形式显示的。

注意:(1) 被乘数和乘数按位相乘时,是从个位到高位逐位计算的,但字符串下标 0 的位置是从高位开始标识的,因此在实现数据读取、结果输出等操作时要注意两者之间

的逆序关系。

(2) 列表是一个定义后内容和长度均可修改的数据类型,但操作时需要注意列表越界问题,即要判断列表下标是否可以正确获取到。

3.2.2 多媒体处理

用 Python 处理多媒体信息并不像想象中的那么复杂,它实际上就是通过直接调用或导入与多媒体数据处理相关的内置函数或第三方库文件,对已经创建好的多媒体对象完成各种操作。

这里将多媒体处理方法简单概括为以下三步:

(1) 导入多媒体内置模块或库文件。

(2) 生成多媒体对象。

(3) 调用多媒体处理函数。

因此,多媒体处理程序编写的难点并不是在于方法的掌握,而是在于如何选取满足实际需求的、正确的、合适的多媒体内置模块、第三方库文件以及相应的处理函数,这就需要多练、多记、多搜,阅读函数的 help 文档是一个不错的方法。

【例 3-10】 实现对图像文件的旋转和缩放等简单处理。

解答: 本题是对图像进行处理,因此选用的是第三方库文件 pillow,具体代码如下:

```
from PIL import Image #导入内置图像处理库
myImage= Image.open("C:\Users\new\Desktop\view1.jpg") #生成图像对象
print(myImage.format, myImage.size, myImage.mode) #读取图片格式、大小和
模式
myImage=myImage.resize((128,128))#通过像素修改图片大小
outImage=myImage.rotate(45) #逆时针旋转 45 度
outImage.show() #显示图像对象
myImage.close() #销毁图像对象
outImage.close() #销毁图像对象
```

【例 3-11】 实现对音频文件的音量控制。



解答：本题是对音频进行处理，在 2.2.2 节中采用内置模块 winsound 创建音频对象，但是，在 winsound 模块中并没有提供音量控制函数。因此，这里选用功能更加丰富的第三方库文件 pygame 来完成对音频文件的复杂处理。其中，pygame 库的 mixer 模块主要实现音效控制等操作。pygame.mixer.music.set_volume()用来控制音量，取值范围为 0～1.0 的浮点数。0 为最小值，1.0 为最大值。

具体代码如下：

```
import pygame.mixer #导入音频处理库的某个模块
pygame.mixer.init() #初始化 mixer 模块
s=pygame.mixer.Sound("ring.wav") #定义音频对象
s.set_volume(0.8) #设置音量
s.play() #播放音频对象
```

说明：(1) pygame 库中的 mixer 模块在使用前需要进行初始化工作，对应语句是 pygame.mixer.init()。

(2) pygame.mixer.Sound()是调用 pygame 库中 mixer 模块的 Sound 函数，用于定义一个音频对象。

(3) s.set_volume()用于设置音频文件的音量，其参数是一个 0～1.0 的浮点数，参数越大，音量越高。

【例 3-12】 抓取视频播放时的画面，并且将其依次存储为图片。

解答：本题是将视频播放的画面依次存储为图片。视频是由一组连续的静态图片按照一定的播放速率形成的动态影像，一帧就是一张图片。因此，要抓取视频中每帧的画面，只需按照视频播放的速度逐一读取各帧内容即可。

```
import cv2 #导入外部视频处理库
myVideo=cv2.VideoCapture("E:/Megamind.avi")#定义视频对象
c=1
timeF=1000 #视频帧播放的速度
if myVideo.isOpened(): #判断视频对象是否能被打开
    while True: #循环播放
```

```
        ret, prevframe=myVideo.read() #视频解码并返回当前帧
        if ret==True: #如果当前帧被成功读取
            cv2.imshow('video', prev)#将当前帧显示在窗口上
    if(c%timeF==0):
        cv2.imwrite('image/'+str(c)+'.jpg',prevframe) #将当前帧存储为
        图像
            c=c+1
        else: #否则
            break #跳出
cv2.destroyAllWindows() #销毁视频播放窗口
```

说明：cv2.imwrite(path, frame)用于将当前图片保存在指定路径中，它一般有两个参数，第一个参数是保存路径和文件名，第二个参数是需要保存的图片对象。

习　题

1. 用 Python 编写程序，输入一个年份，判断该年是不是闰年并且输出结果。

注：凡符合下面两个条件之一的年份是闰年：①能被 4 整除但不能被 100 整除；②能被 400 整除。

2. 设计学生信息录入程序(从 Python 的 Shell 界面输入和显示信息)。具体要求如下：

(1) 8 位纯数字学号。

(2) 18 位的身份证号，并从身份证号中自动获取出生年月日。

(3) 显示大写的英文姓名。

(4) 家庭住址，限制在 20 个汉字以内，越界的报错。

(5) 性别，如果性别输入的是 Girl、Boy 或其他内容，均修改为女或男。

“学生信息录入”程序实例如图 3-3 所示。



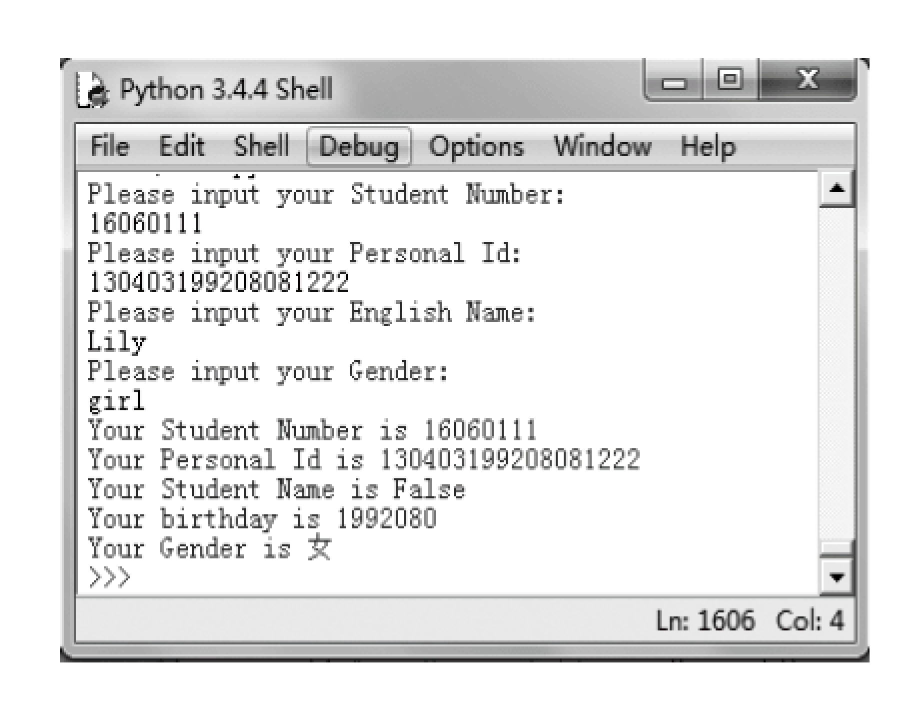


图 3-3　“学生信息录入”程序实例

3. 如何使用信用卡最划算？信用卡有两种还款方式，分别是最低还款额和分期还款额（某银行人民币存款利率如表 3-9 所示）。

表 3-9　某银行人民币存款利率表

项　　目		年利率/%
活期	0.35	
定期	三个月	1.35
	半年	1.55
	一年	1.75
	二年	2.25
	三年	2.75
	五年	2.75

4. 设计一个字符串加密程序，使其可以对输入的明文（字符串形式）进行加密，同时还可以对已经加密的密文进行解密，解密结果应该与输入的明文一致。

5. 输入一个字符串，分别统计其中的数字、字母和标点符号的个数。

6. 统计输入的字符串中的单词个数、最长单词长度等。

7. 实现一个 strcmp()函数。

8. 判断输入的字符串是否为回文（无论正方向读还是反方向读均为同一字符串），如果是回文则输出"Yes."，如果不是回文则以相反的顺序输出原字符串。

9. 输入一个字符串，将驼峰命名法字符串转换为下画线命名法字符串。

10. 用多种方法实现对字符串从任意位置开始的任意长度的截取。



第 4 章　计算机的流程控制

简单的计算用很少的步骤就可以实现，如计算圆的周长和面积的程序，只需要按照“接收用户输入的半径→计算周长→计算面积→输出结果”的步骤顺序执行即可实现。然而，有些问题用这种“一条道走到头”的方法是无法实现的。试想一下小孩经常玩儿的猜数字游戏，A 手里写上某个数字(如 6)，B 猜这个数字是几。当 B 说出一个数字后，A 说出“对”或“不对”的结果。这就需要一个判断过程：B 说出的数字是否与 A 手中的数字相等，A 根据判断的结果或者说“对”或者说“不对”。如果我们编写一个程序来实现这个功能，就需要借助条件判断语句，根据判断条件来决定输出的结果。还可以让这个游戏更有趣一些，当 B 说出数字时，A 不直接说出“对”或“不对”，而是不断提示 B，直到他猜对为止，这就需要重复执行猜数字的过程。这两种情况都使得程序不再是按顺序执行的了。

本章介绍计算机程序的三种控制结构：顺序结构、选择结构和循环结构，这三种控制结构是组成更复杂的程序的基础。通过案例的分析，学习如何在 Python 中实现计算机的流程控制。

4.1　计算机的逻辑

从原理角度，计算机是一个逻辑处理器。计算机所做的一切工作最后都是被转化成逻辑运算来执行的。



4.1.1　逻辑表达式

计算机既能进行算术运算,也能进行逻辑运算。从前面的实例中我们已经了解到,在条件语句和循环语句中都要用到条件判断。接下来以“由简到繁”角度来介绍条件语句的构成。

1. 逻辑值

逻辑值,也称布尔值,是最简单的条件判断语句,即 True 或 False。

例如:

```
>>>a=20
>>>a>40
```

结果是 False。

再如:

```
>>>b=-90
>>>b+50==100
```

结果是 False。

如下语句:

```
>>>while True:
    print("Print forever!")
```

因为该程序的条件判断语句恒为真,它将无限输出"Print forever!"语句。

2. 关系表达式

关系表达式是用关系运算符将两个能计算出逻辑值(True 或 False)的表达式(如算术表达式、关系表达式、逻辑表达式、赋值表达式、字符表达式等)连接起来的式子。

例如:

```
if a<b:
    print("a is smaller.")
```

首先，比较 a 和 b 的值，当 a 小于 b 的结果是“真”(即是 True)时，则执行 print()语句。

这里，“a<b”是关系表达式，执行关系运算，“<”是关系运算符。关系运算是比较简单的逻辑运算，它用来比较关系运算符两边的变量并且得出一个逻辑值。Python 中的关系运算符如表 3-5 所示。

3. 逻辑表达式

进一步地，用逻辑运算符将关系表达式或逻辑值连接起来的有意义的式子称为逻辑表达式。例如：

```
if a<b and a<c:
    print(" a is the smallest one.")
```

这里，“a<b and a<c”是逻辑表达式，表示只有当“a<b”和“a<c”两个关系表达式的值均为“真”(即是 True)时，才执行 print()语句。and 是逻辑运算符，它将两个关系表达式连接起来形成逻辑表达式。Python 提供三种逻辑运算符，分别是“与”(and)、“或”(or)和“非”(not)，如表 3-6 所示。

4.1.2 运算符优先级

关系表达式和逻辑表达式一般都是由能够计算出逻辑值的多个表达式组成的。那么，这多个表达式的计算顺序是什么？这就涉及运算符优先级的问题。解决该问题之前，首先需要了解构成表达式的运算符的种类。

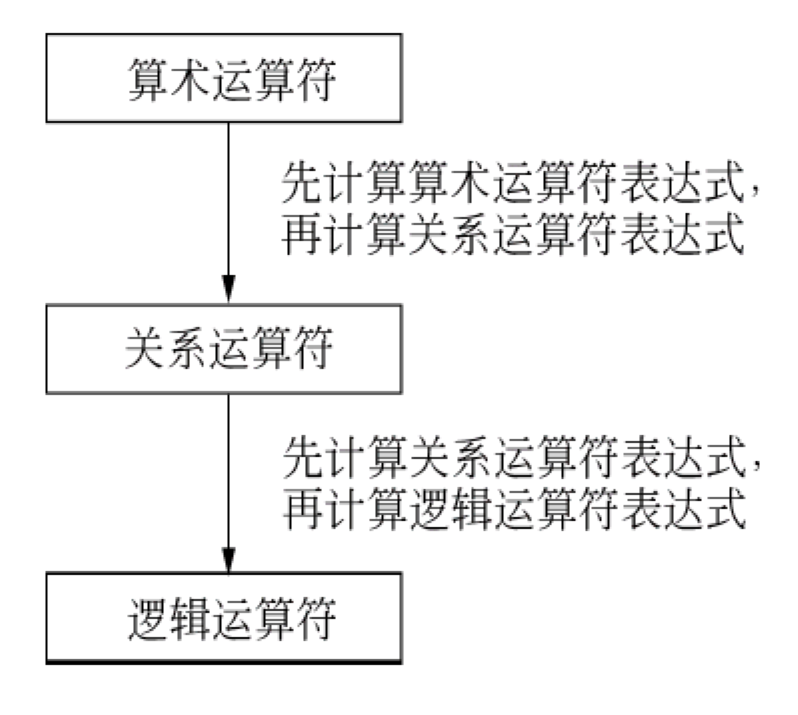


图 4-1 运算符的优先级

计算机能够处理算术运算和逻辑运算。其中，算术运算使用算术运算符(如＋、－、*、/等，详见表 3-1)，而逻辑运算使用关系运算符(如＜、＜＝、＞、＞＝、＝＝、!＝、is、is not 等，详见表 3-5)和逻辑运算符(如 and、or、not 等，详见表 3-6)。因此，在计算时，需要判断表达式中算术运算符、关系运算符和逻辑运算符三者之间的优先级关系。



算术运算符、关系运算符和逻辑运算符的优先级关系为：算术运算符＞关系运算符＞逻辑运算符，如图 4-1 所示。进一步地，三个逻辑运算符的优先级关系为：not＞and＞or。注意，可以通过添加括号的方式来改变表达式中运算符间的优先级。表 4-1 列举了几个典型的逻辑表达式，并描述了它们的具体执行过程。

表 4-1　逻辑表达式实例及其执行过程

执行步骤	逻辑表达式			
	a＜b and b＞c	**a＜b or a＜c**	**not a＜b**	**a＜b or a＜c and b＜c**
第一步	True and b＞c	True or a＜c	not True	True or a＜c and b＜c
第二步	True and True	True or False	False	True or False and b＜c
第三步	True	True	—	True or False and False
第四步	—	—	—	True or False
第五步	—	—	—	True

4.2　程序的有序执行

1. 顺序结构

顺序结构是指程序的功能是通过从头到尾依次执行各条语句来实现的。

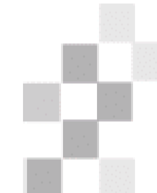
【例 4-1】　根据三角形的三条边长计算三角形面积。

解答：由海伦公式知道，三角形的面积 $S=\sqrt{p(p-a)(p-b)(p-c)}$。其中，p 为周长的一半，即 $p=(a+b+c)/2$。

经过分析可知，只要采用“接受用户输入、转换数据类型、计算周长、计算面积和输出结果”等几个顺序执行的步骤，就可以实现计算三角形周长和面积的功能。因此，本程序采用简单的顺序结构。

具体代码如下：

```
#输入三角形的三条边,计算三角形的面积
```




```
import math #引入 math 模块
a=input("请输入第一条边的长度:")
b=input("请输入第二条边的长度:")
c=input("请输入第三条边的长度:")
a=float(a) #将 a 强制类型转换成浮点数
b=float(b)
c=float(c)
p=(a+b+c)/2 #计算周长的一半
s=math.sqrt(p*(p-a)*(p-b)*(p-c)) #计算面积
print("三角形面积为:",s)
```

从上述代码可以看出,程序从入口处(函数外的第一条语句)开始执行,通过顺序执行各条语句,到出口处(最后一条语句)结束执行。

2. 选择结构

在执行例 4-1 的程序时发现,如果用户输入了非正常数据(如输入了英文字母),则程序运行时会崩溃并抛出错误提示。为了避免出现这样的情况,需要增加一个条件判断语句:如果用户输入了非正常数据,则给出相应的提示信息并结束程序的运行。这就使得语句的执行需要根据条件判断的结果来做出选择。如图 4-2 所示,当条件表达式的值为“真”(即是 True)时,程序执行语句序列 1;当条件表达式的值为“假”(即是 False)时,程序执行语句序列 2。

Python 中用 if 语句实现程序的条件结构。

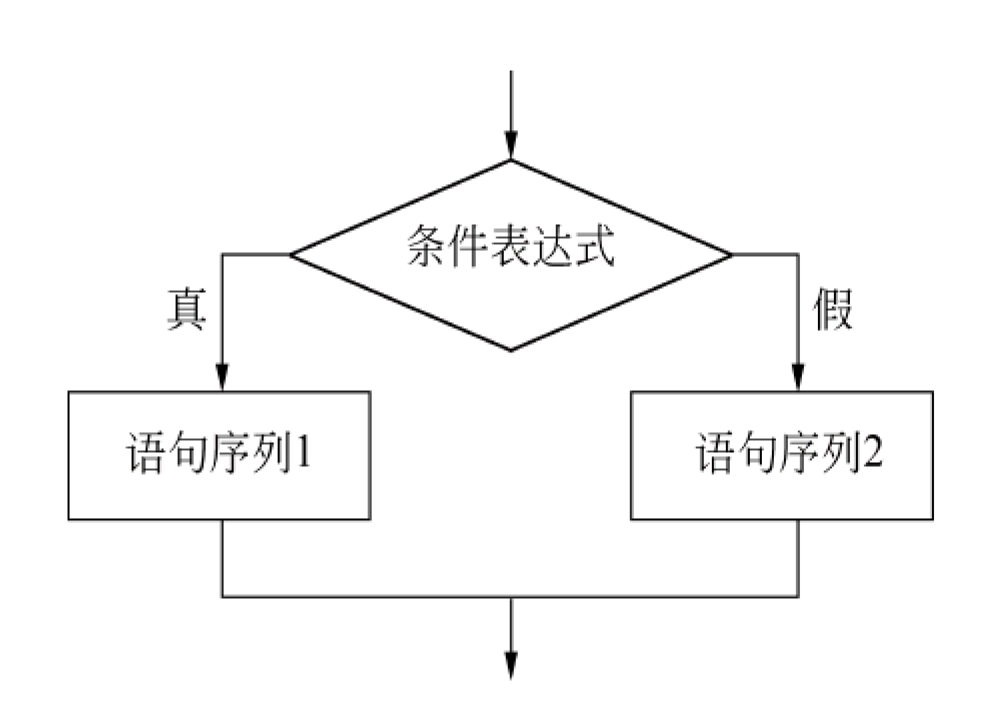


图 4-2　选择结构流程图



3. 循环结构

在猜数字的游戏中，如果让 A 不断地提示 B，则 B 猜的数字最终会与 A 给出的数字相等。显然，这是个不断重复“猜—提示—再猜”的过程。在计算机程序中，可以使用循环语句来表示重复执行的过程。如图 4-3 所示，当条件表达式的值为“真”(即是 True)时，程序执行循环体内的语句序列，执行完毕则返回条件表达式处，重新判断循环条件。一旦条件表达式的值为“假”(即是 False)时，则程序结束循环，并跳出循环体继续执行后面的语句。

Python 中使用 while 和 for 语句实现程序的循环结构。

注意：while 语句一般用于循环次数不确定的情况，而 for 语句用于明确知道循环体执行次数的情况。

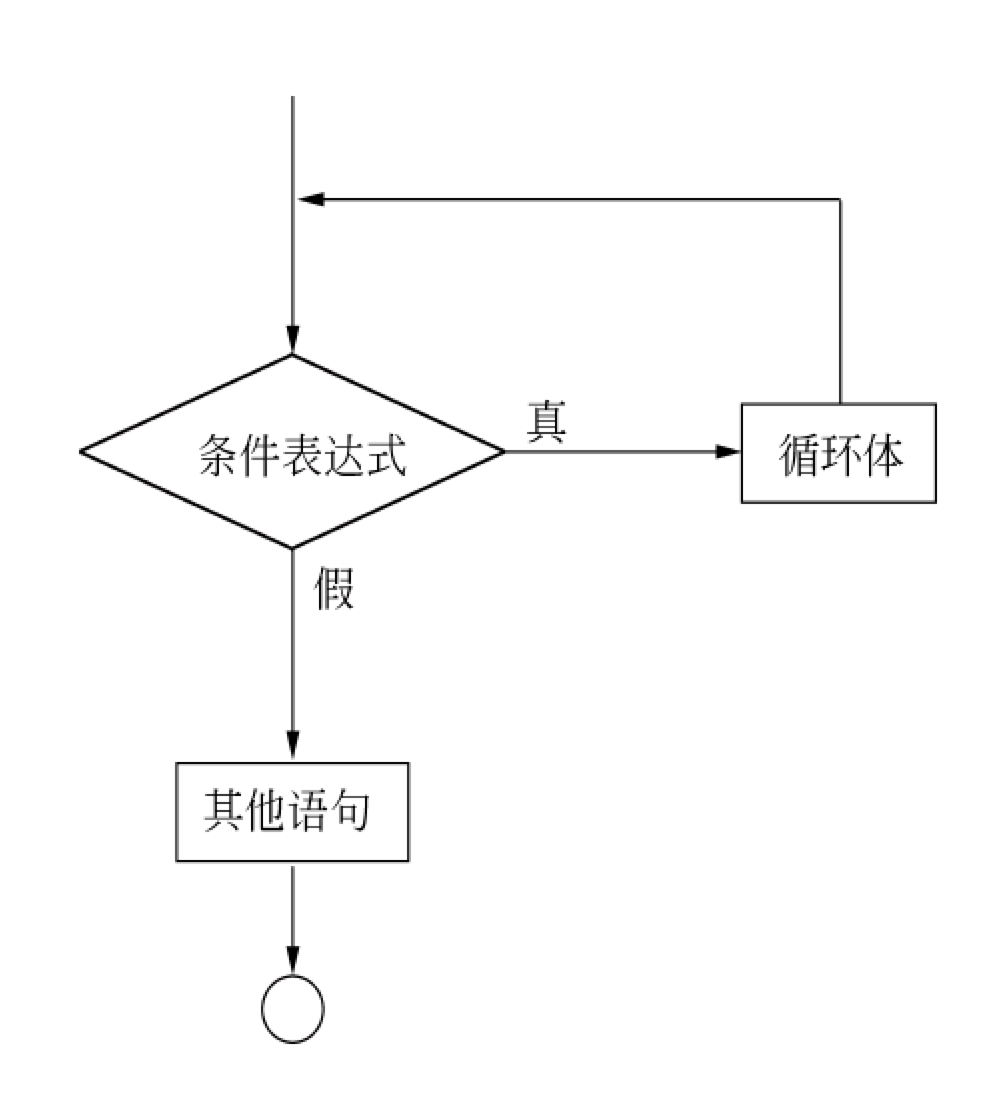


图 4-3　循环结构流程图

4.2.1　if 条件语句

1. 单分支条件语句

单分支条件语句是通过判断一个条件表达式的值来选择执行或不执行某个语句序列。它的执行过程如图 4-4 所示，当条件表达式的值为“真”(即是 True)时，执行语句序列 1，否则跳过语句序列 1，直接执行条件表达式后面的其他语句。

单分支 if 条件语句的格式如下(注意缩进和冒号):

```
if <条件表达式>:
    语句序列
其他语句
```

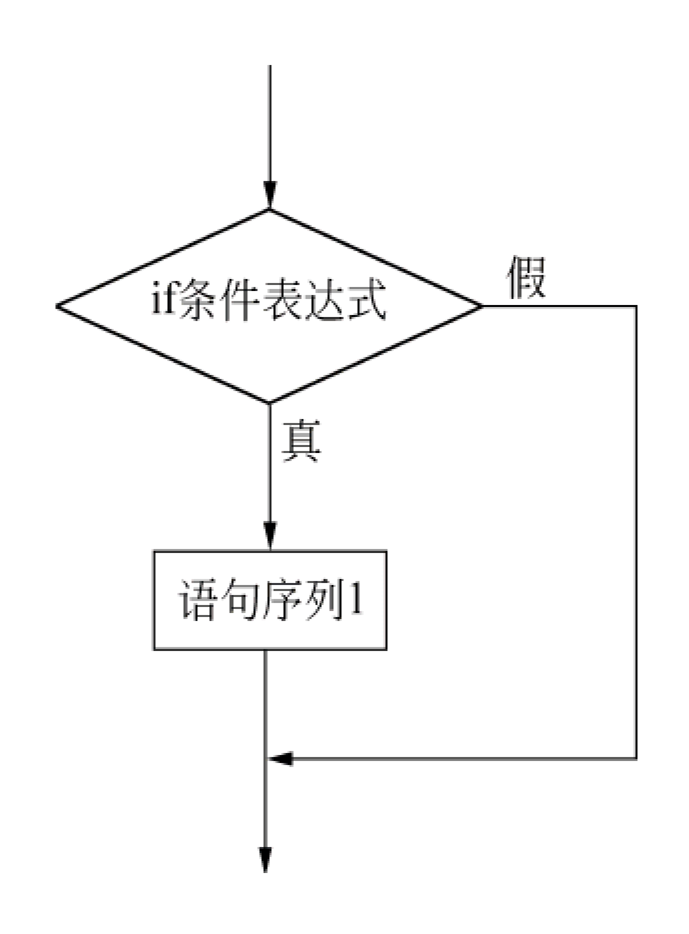


图 4-4　单分支 if 条件语句流程图

【例 4-2】 猜字游戏,已知正确数字是 5(用单分支 if 语句实现)。

解答:程序需要用户从控制台输入一个数字,然后判断该数字是否是 5。如果是 5,则输出"猜测正确";否则,则输出"猜测失败"。

具体代码如下:

```
print("Welcome!")
g=input("Guess the number:") #input 函数用于从控制台输入内容,返回 string 类型
guess=int(g) #将 string 类型转换为 int 类型
num=5
if guess==num: #判断语句
    print("You win!")
if guess>num or guess <num:
    print("You lose!")
print("Game over!")
```

思考:(1) 当猜测正确时,即 guess=5,并且 if guess==num 语句中条件表达式的值为"真"(即是 True)时,后面的 if guess>num or guess<num 语句还会被执行吗?根据一般情况下程序是顺序执行的特点,无论前面的 if 语句是否满足判断条件,其后的各条 if 语句都会被执行。因此,由多条单分支 if 条件语句构成的程序,其执行效率通常比较低。

(2) 如果将程序中的 num 直接用常量 5 代替可以吗?请读者回顾变量的作用。



2. 双分支条件语句

双分支条件语句是通过判断一个条件表达式的值，来选择执行语句序列 1 还是执行语句序列 2。它的执行过程如图 4-5 所示，当条件表达式的值为“真”(即是 True)时，执行语句序列 1；否则，执行语句序列 2，然后再继续执行双分支条件语句后面的其他语句。双分支 if 条件语句的格式如下(注意缩进和冒号)：

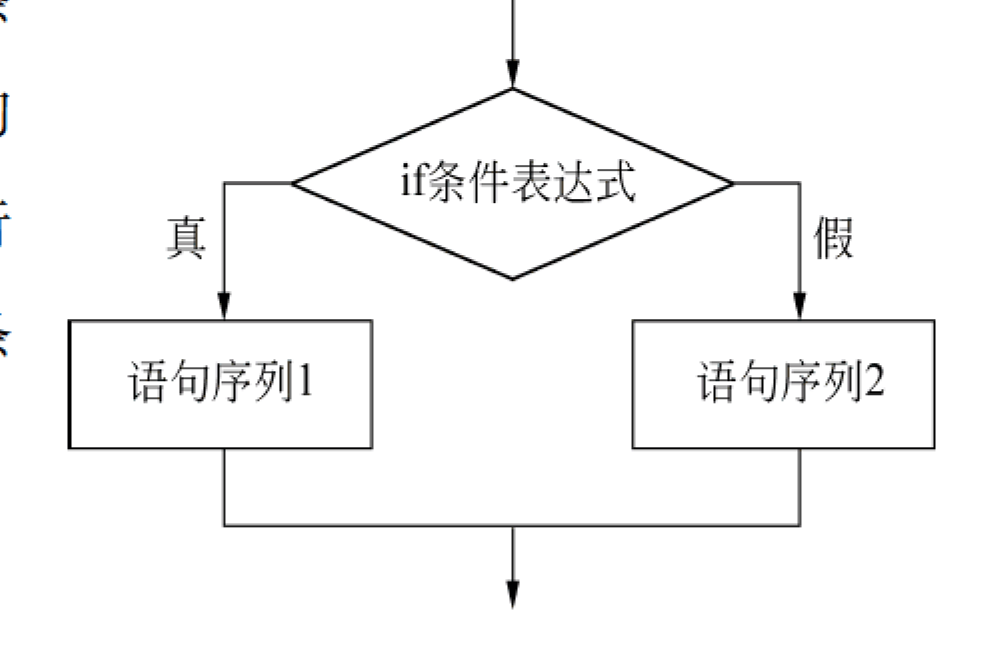


图 4-5　双分支 if 条件语句流程图

```
if <条件表达式>:
    语句序列 1
else:
    语句序列 2
其他语句
```

为了提高程序的执行效率，对例 4-2 中的程序进行适当修改。请读者分别运行例 4-2 和例 4-3 两个程序，并比较它们的执行过程。

【例 4-3】 改进例 4-2，用双分支 if-else 语句实现。

解答：提高程序效率的一个直接且有效的方法就是尽量减少执行的代码量。针对例 4-2 中多条单分支 if 语句间存在冗余执行的问题，我们希望当用户猜测正确后就结束游戏，无须再做无用判断。

具体代码如下：

```
print("Welcome!")
g=input("Guess the number:")#input 函数用于从控制台输入内容，返回 string
类型
guess=int(g) #将 string 类型转换为 int 类型
num=5
if guess==num: #判断语句
    print("You win!")
else : #将例 4-2 中的 if 修改为 else
```




```
    print("You lose!")
print("Game over!")
```

说明：与例 4-2 相比，例 4-3 的执行效率较高。因为当 if guess==num 中条件表达式的结果为“真”(True)时，程序将直接跳转到最后的 print()语句，不会再执行 else 中的语句。

3. 多分支条件语句

如果需要判断的条件比较复杂，则需要设计多分支的 if 条件语句。图 4-6 表示的是三层分支条件语句。当条件表达式 1 为“真”(True)时，则执行语句序列 1；否则，再判断条件表达式 2，当条件表达式 2 为“真”(True)时，则执行语句序列 2；否则，再判断条件表达式 3，当条件表达式 3 为“真”(True)时，执行语句序列 3；否则，继续执行多路分支条件语句后面的语句。多路分支条件语句是通过使用 if 语句的嵌套来实现的。

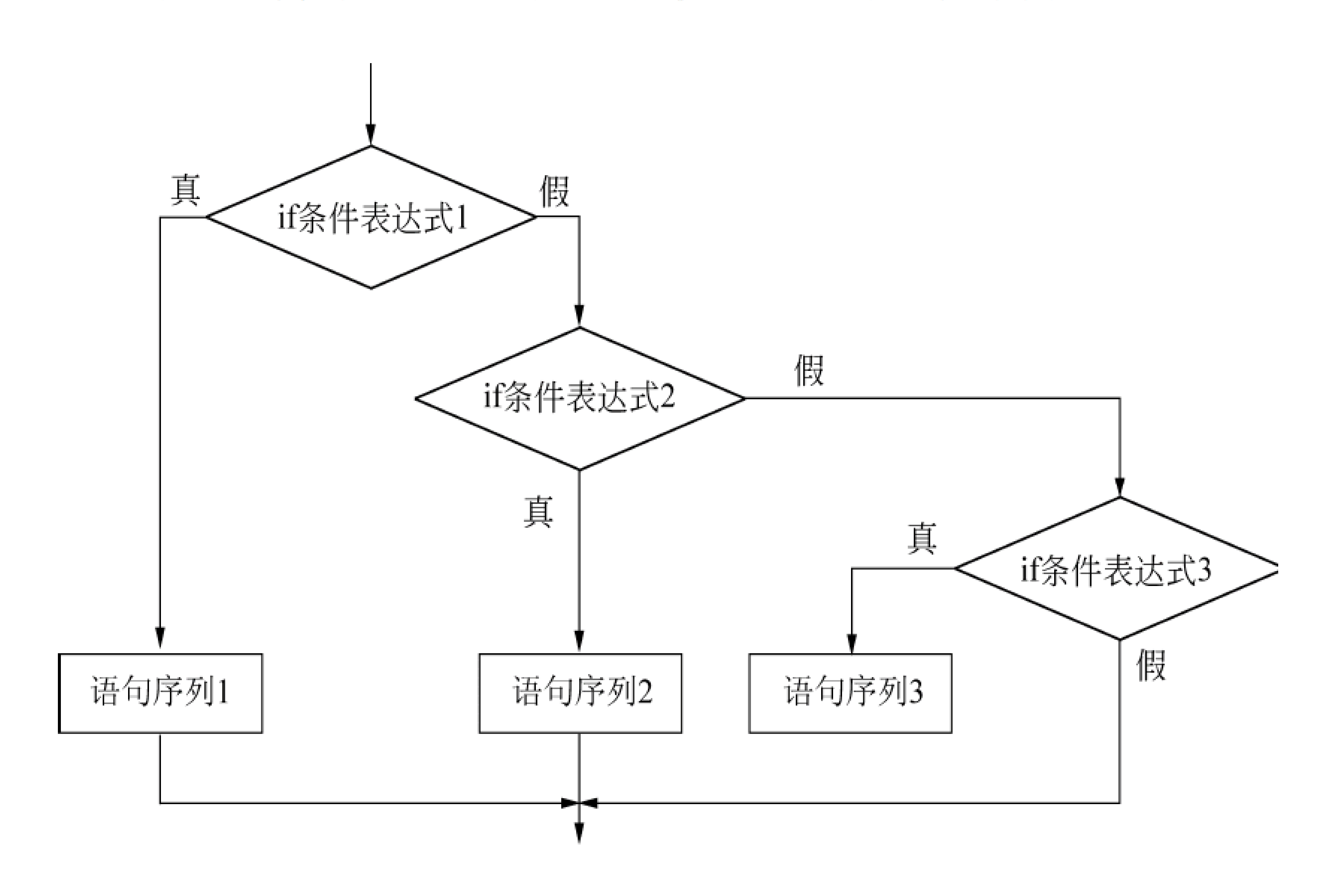


图 4-6　多分支 if 条件语句流程图

【例 4-4】 继续改进例 4-3，要求对于猜错的数字，要提示其错误原因。

解答：程序需要用户从控制台输入一个数字，然后判断该数字是否是 5。如果是 5，则输出猜测正确；否则，继续判断错误数字和 5 的大小关系，并且输出相应的错误提示。这涉及多个相关条件的级联判断，因此，需要用到 if 语句的嵌套。



具体代码如下：

```
print("Welcome!")
g=input("Guess the number:") #input 函数用于从控制台输入内容,返回 string
类型
guess=int(g) #将 string 类型转换为 int 类型
num=5
if guess==num: #判断语句
    print("You win!")
else:
    if guess>5:
        print("larger than 5!")
    else:
        print("smaller than 5!")
print("Game over!")
```

【例 4-5】 输入三条边,判断是否能组成直角三角形。

解答：输入任意三条边的值,如何判定是否能组成直角三角形？从以前学过的数学知识知道,如果两边之和大于第三边,则可以构成三角形。进一步地,如果满足勾股定理的三角形,则是直角三角形。我们可以通过绘制流程图的方式,将上面的数学语言翻译成计算机所能理解的条件判断过程,如图 4-7 所示。另外,流程图也可以帮助我们更好地

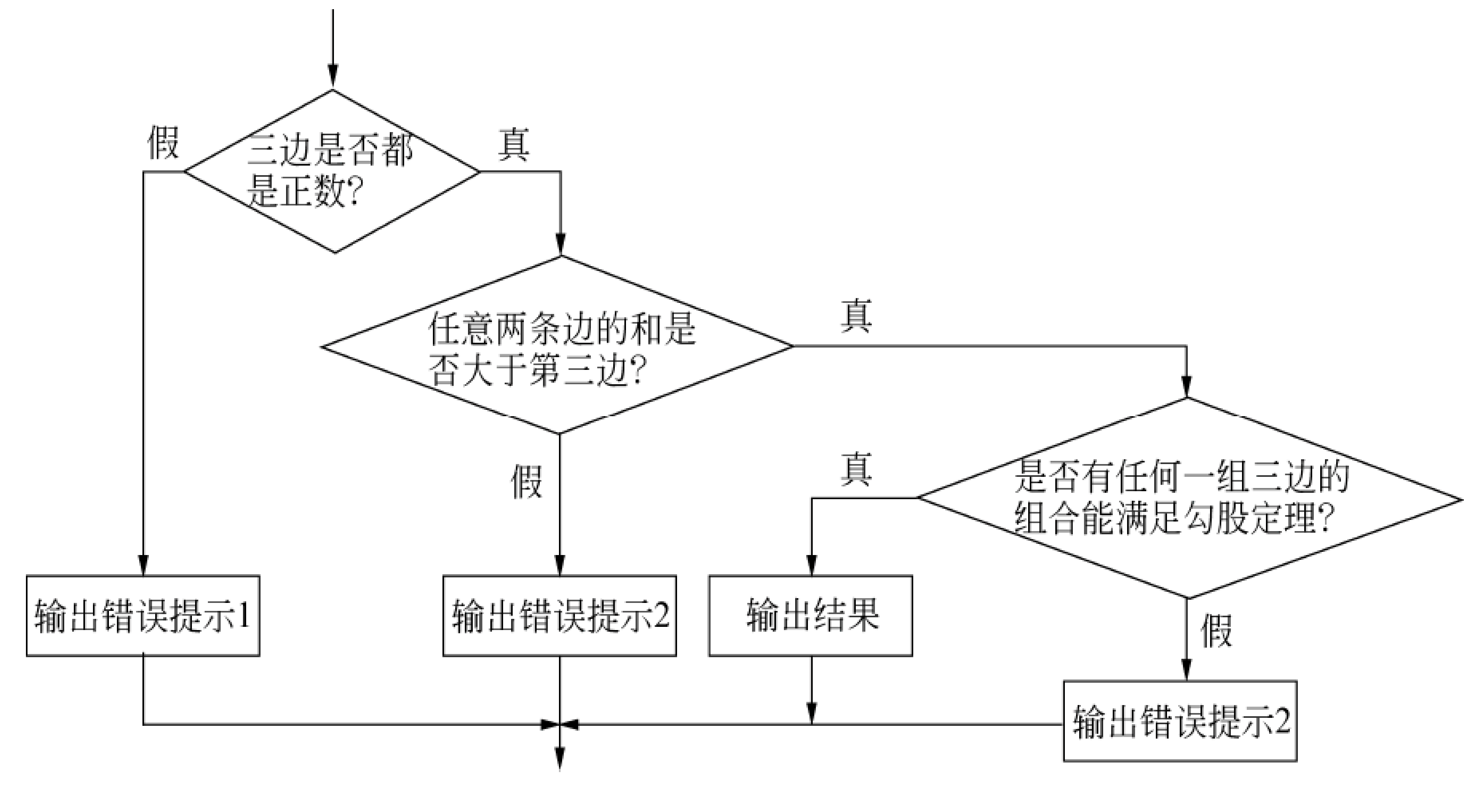


图 4-7　例 4-4 条件判断流程图

分析和编写程序。

具体代码如下：

```
a=float(input("input the first number:"))#用 input()函数接收从键盘输入的
数据,并将默认的字符串类型转换成 float()类型
b=float(input("input the second number:"))
c=float(input("input the third number:"))
#如果输入为 0 则问题无意义
if a >0 and b >0 and c >0:
#如果任意两条边的和小于等于第三边则不能构成三角形
    if a+b >c or a+c >b or b+c >a :
    #如果满足勾股定理则为直角三角形
        if a * a+b * b==c * c or a * a+c * c==b * b or b * b+c * c==a * a:
            print("该三条边可以组成直角三角形!")
        else:
            print("该三条边可以组成普通三角形!")
    else:
         print("输入的三条边无法构成三角形!")
else:
    print("输入错误!")
```

4. 复合条件语句

例 4-5 使用了三层 if 嵌套语句,如果是更加复杂的条件判断问题,则嵌套的深度会更深,这将使程序变得难以设计和阅读。为了解决这个问题,Python 提供了 if-elif-else 的复合结构。该结构将成对出现的 if 和 else 合并成了一个 elif 子句。if-elif-else 格式的条件语句常用于替换多路分支的 if 条件语句,使得程序结构更加清晰。它也属于 if 语句的嵌套。

if-elif-else 的格式如下：

```
if <条件表达式 1>:
    语句序列 1
```



```
elif<条件表达式 2>:
    语句序列 2
elif<条件表达式 3>:
    语句序列 3
    …
else:
    语句序列 n
其他语句
```

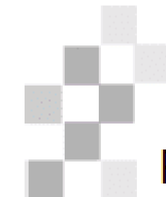
【例 4-6】　用 if-elif 复合结构修改例 4-5 程序。

解答：可以用 if-elif-else 结构替换例 4-5 中的三个 if-else 结构来实现“输入三条边，判断是否是直角三角形”的功能。请比较例 4-5 和例 4-6 给出的代码，找出它们在逻辑结构和条件表达式上的异同。

```
#输入三条边，判断是否组成直角三角形
a=float(input("input the first number:"))
b=float(input("input the second number:"))
c=float(input("input the third number:"))
if a<=0 or b<=0 or c<=0:
    print("输入错误!")
elif a+b <=c or a+c <=b or b+c <=a :
    print("这三条边不能组成三角形!")
elif a*a+b*b==c*c or a*a+c*c==b*b or b*b+c*c==a*a:
    print("该三条边可以组成直角三角形!")
else:
    print("该三条边可以组成普通三角形!")
```

说明：例 4-6 和例 4-5 中的条件判断语句虽然使用了不同的条件表达式，但它们却能实现相同的功能。由此可知，同样的题目有多种多样的解决方法。

从例 4-6 的代码中可以看出，使用 if-elif-else 格式可以像嵌套的 if 语句一样实现多路分支，但其并列的格式却使程序看起来整齐得多，增加了代码的可读性。if-elif-else 语



句顺序判断每一个条件表达式，找到第一个为真（True）的条件，就执行其冒号“：”后面缩进的语句体，执行完毕后直接跳转到 if-elif-else 后面的语句序列继续执行。

【例 4-7】 输入考生的考试成绩，输出其所在的成绩等级。

解答：本实例需要实现“任意给出一个考试分数，判定其成绩等级”的功能。如果成绩按照优、良、中、及格、不及格 5 个等级来划分（≥90 分为优、80～89 分为良、70～79 分为中、60～69 分为及格、<60 分为不及格），那么需要判定 4 个条件。

具体代码如下：

```
#输入分数，评判成绩等级
score=float(input("input the score:"))
if score>=90:
    grade="优"
else:
    if score>=80 and score<90:
        grade="良"
    else:
        if score>=70 and score<80:
            grade="中"
        else:
            if score>=60 and score<70:
                grade="及格"
            else:
                grade="不及格"
print("The grade is :",grade)
```

例 4-7 给出了使用 if 语句嵌套实现的代码，请读者试着改为使用 if-elif-else 格式的程序。

通过单分支、双分支和多分支等不同类型的 if 条件语句的学习，我们需要深入理解 if 语句和 if-else 语句的异同点：

（1）多条单分支 if 语句中，各个条件表达式之间是并列关系，它们相互独立。

句顺序判断每一个条件表达式,找到第一个为真(True)的条件,就执行其冒号“:”后面缩进的语句体,执行完毕后直接跳转到 if-elif-else 后面的语句序列继续执行。

【例 4-7】 输入考生的考试成绩,输出其所在的成绩等级。

解答:本实例需要实现“任意给出一个考试分数,判定其成绩等级”的功能。如果成绩按照优、良、中、及格、不及格 5 个等级来划分(≥90 分为优、80～89 分为良、70～79 分为中、60～69 分为及格、<60 分为不及格),那么需要判定 4 个条件。

具体代码如下:

```
#输入分数,评判成绩等级
score=float(input("input the score:"))
if score>=90:
    grade="优"
else:
    if score>=80 and score<90:
        grade="良"
    else:
        if score>=70 and score<80:
            grade="中"
        else:
            if score>=60 and score<70:
                grade="及格"
            else:
                grade="不及格"
print("The grade is :",grade)
```

例 4-7 给出了使用 if 语句嵌套实现的代码,请读者试着改为使用 if-elif-else 格式的程序。

通过单分支、双分支和多分支等不同类型的 if 条件语句的学习,我们需要深入理解 if 语句和 if-else 语句的异同点:

(1) 多条单分支 if 语句中,各个条件表达式之间是并列关系,它们相互独立。



（2）成对的 if-else 语句中，各个条件表达式之间是递进关系，它们相互影响。

（3）在 if-else 结构中，如果 if 语句的条件表达式的值为“真”（即是 True）时，则 else 中的语句不会被执行。如果存在多个 if 语句，则各 if 语句均要被执行。

4.2.2　while 循环语句

while 语句一般用于循环次数不确定的情况下，其格式如下：

```
while <条件表达式>:
    语句序列
其他语句
```

当条件表达式的值为“真”（即是 True）时，则执行循环体中的语句序列。执行完毕后，返回 while 语句的条件表达式位置，再次判断循环条件的值。while 语句重复执行“检查循环条件—执行循环体—返回循环条件”的过程，直到条件表达式的值为“假”（即是 False）时，则结束并跳出循环，转而执行循环体后面的语句。

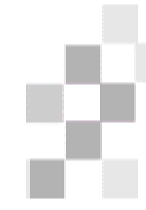
【例 4-8】　计算 1～n 的累加和。

解答：计算 1～n 的累加和的过程是一个重复执行加法运算的过程，且重复执行的次数是确定且已知的，因此，可以使用 while 语句实现。

具体代码如下：

```
n=int(input("请输入大于 0 的自然数 n:"))
sum=0
i=0 #循环控制变量
while i<=n:
    sum=sum+i
    i=i+1
print("1~",n,"的累加和是:",sum)
```

说明：循环控制变量是指用在循环语句的条件表达式中，用来控制循环次数和循环的执行过程的变量。



此程序执行循环体的条件是用户在键盘上输入的自然数 n 大于 0。如果条件表达式 i<=n 的值是“真”(True)，则执行循环体。循环体中有一个 i=i+1 的语句，这是对循环控制变量 i 的控制。试想如果没有这条语句会怎么样呢？如果没有这条语句，程序就会一直重复执行循环体中的语句，永远不会结束，即进入了“死循环”。所以，在循环体中必须有一个对循环控制变量的递增或递减操作，以最终达到条件表达式的值为“假”(False)时而结束循环。

运行例 4-8 的代码可以发现，如果用户输入了小于 0 的数，则循环体一次都不会执行。但是，在有些情况下，程序需要循环体必须先执行一次，然后再判断循环条件。例如，在登录系统中，需要验证用户名和密码是否正确，如果不正确，则提示错误信息并让用户重新输入，直至输入正确为止。与例 4-8 中的“先判断循环条件再执行循环体”的执行过程比较，这种情况是“先执行循环体再判断循环条件”。在有的编程语言中有专门的语句来对应“先判断”和“后判断”两种情况，例如，在 C 语言中分别提供了“当型”循环语句 while 和“直到型”循环语句 do-while。在 Python 语言中，可以通过设置合适的循环控制变量初始值，使得循环语句的条件表达式的值一开始就为 True，这样循环体至少执行一次，然后在循环体中根据实际需要改变循环控制变量的值，使之变为 False 从而结束循环。请读者试着分析例 4-9 猜数游戏程序中的代码，体会实现“直到型”循环的方法。

【例 4-9】 继续改进例 4-4 中的猜数游戏。要求当用户猜测错误时，可以允许其反复输入。

解答：由于用户猜字过程是重复的且猜测次数不知，因此，适合使用 while 循环语句。又因为在猜数游戏中，用户必须有第一次的输入，游戏才能开始。因此，本例属于“直到型”的 while 循环语句。但是，对于“直到型”的 while 循环语句，如何跳出 while 循环呢？一般有两种实现方法。

(1) 方法一：使用循环控制变量控制循环次数。

该方法通过在循环执行过程中改变循环控制变量的值，来控制循环的执行或跳出。循环控制变量就是控制循环执行的开关。

具体代码如下：



```
answer=0 #定义循环控制变量
    while answer==0: #循环语句
        print("Welcome!")
        g=input("Guess the number:")
        guess=int(g)
        if guess==5:
                answer=1
                print("You win!")
                print("Game over!")
        else:
                if guess >5:
                        print("Too high!")
                else:
                        print("Too low!")
```

（2）方法二：使用 break 语句强制跳出循环。

该方法无须定义循环控制变量。执行时，while 循环中条件表达式的值被设置为恒真，即 while 循环被无条件地执行。当需要结束循环时，它通过使用 break 语句强制跳出循环。

具体代码如下：

```
print("Welcome!")
while True:  #循环语句
    g=input("Guess the number:")
    guess=int(g)
    if guess==5:
        print("You win!")
        print("Game over!")
        break #强制跳出循环
    else:
        if guess >5:
            print("Too high!")
```

```
        else:
            print("Too low!")
```

这里需要注意,如果直接将 while 语句中条件表达式的值设置为 True 或 1(恒真),则循环体会被一直执行,即程序进入“死循环”状态。除非有特殊需要,一般要避免程序出现死循环的情况。

解决 while 语句出现死循环的方法主要有两种:

(1) 避免将 while 语句中条件表达式的值直接设置为 True 或 1。

(2) 在循环体中使用 if 条件语句对循环结束条件进行判断,并配套使用 break 语句跳出循环。

【例 4-10】 依次输入学生各科成绩,计算其总分。程序以“数字 0 或任意字母”为结束标识。

解答:本题中,学生输入各科成绩的次数未知,因此,选择 while 语句实现计算总分的操作。

具体代码如下:

```
sum=0
while True:
    score=input("请输入学生成绩:")
#如果输入成绩为 0 或任意字母,则用 break 跳出循环
    if score=='0' or score>='a' and score<='z' or score>='A' and score<=
'Z':
        break #强制结束循环
    sum=sum+float(score)
print("总分等于",sum)
```

while 常用于解决循环次数未知的问题。当然,它也可以解决循环次数确定的问题,且一般通过循环控制变量控制循环次数。

【例 4-11】 求 1～100 所有偶数的和。

解答:本例中可以通过给循环控制变量设置合适的值来精准控制循环次数。



```
sum=0
i=1
while i<101:
    if i%2==0:
        sum=sum+i
    i+=1 #使用了复合运算符,等同于 i=i+1
print("1~100 所有偶数之和是:",sum)
```

4.2.3　for 循环语句

for 循环语句只能解决具有确定循环次数的循环问题。for 语句的一般格式如下：

```
for<循环控制变量>in <可遍历表达式>:
    <循环体>
```

for 语句中的可遍历表达式通常是字符串、列表、元组、字典和集合等具有确定成员个数的可迭代对象(可迭代对象是指可以直接作用于 for 循环的对象)。for 语句中的循环控制变量用来逐一读取可遍历表达式中各成员的值。for 语句中的关键字“in”用于判断循环控制变量是否在可遍历表达式中,如果在,则将可遍历表达式中的当前值赋值给循环控制变量,并执行循环体;否则,结束循环,继续执行 for 语句后的指令。注意,可迭代对象可以在 for 语句中被直接遍历,即 for 语句会自动地将可遍历表达式中的各成员逐一赋值给循环控制变量,不需要额外编写代码。

for 循环语句流程图如图 4-8 所示。

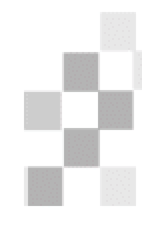
for 语句中循环控制变量的值主要有两种赋值方法：

(1) 赋值为迭代器中各成员的索引,此时,可遍历表达式是由 range()函数表示的集合数据类型的变量的索引范围。

(2) 赋值为迭代器中各成员的值,此时,可遍历表达式是集合数据类型的变量自身。

for 语句与 while 语句虽然都是实现循环操作,但它们存在明显差异,主要体现在：

(1) for 语句中的循环次数必须是已知的,而 while 语句对循环次数没有严格限制。



(2) for 语句中循环控制变量的不断赋值过程是由 for 语句自动实现的,而 while 语句中循环控制变量的修改必须通过编写特定的代码实现。

说明:迭代器是对象要求支持迭代器协议的对象,在 Python 中,支持迭代器协议就是实现对象的__iter__()和 next()方法。其中,__iter__()方法返回迭代器对象本身;next()方法返回容器的下一个元素,在结尾时引发 StopIteration 异常。正是因为使用了 next()方法,for 语句才可以直接遍历迭代器。

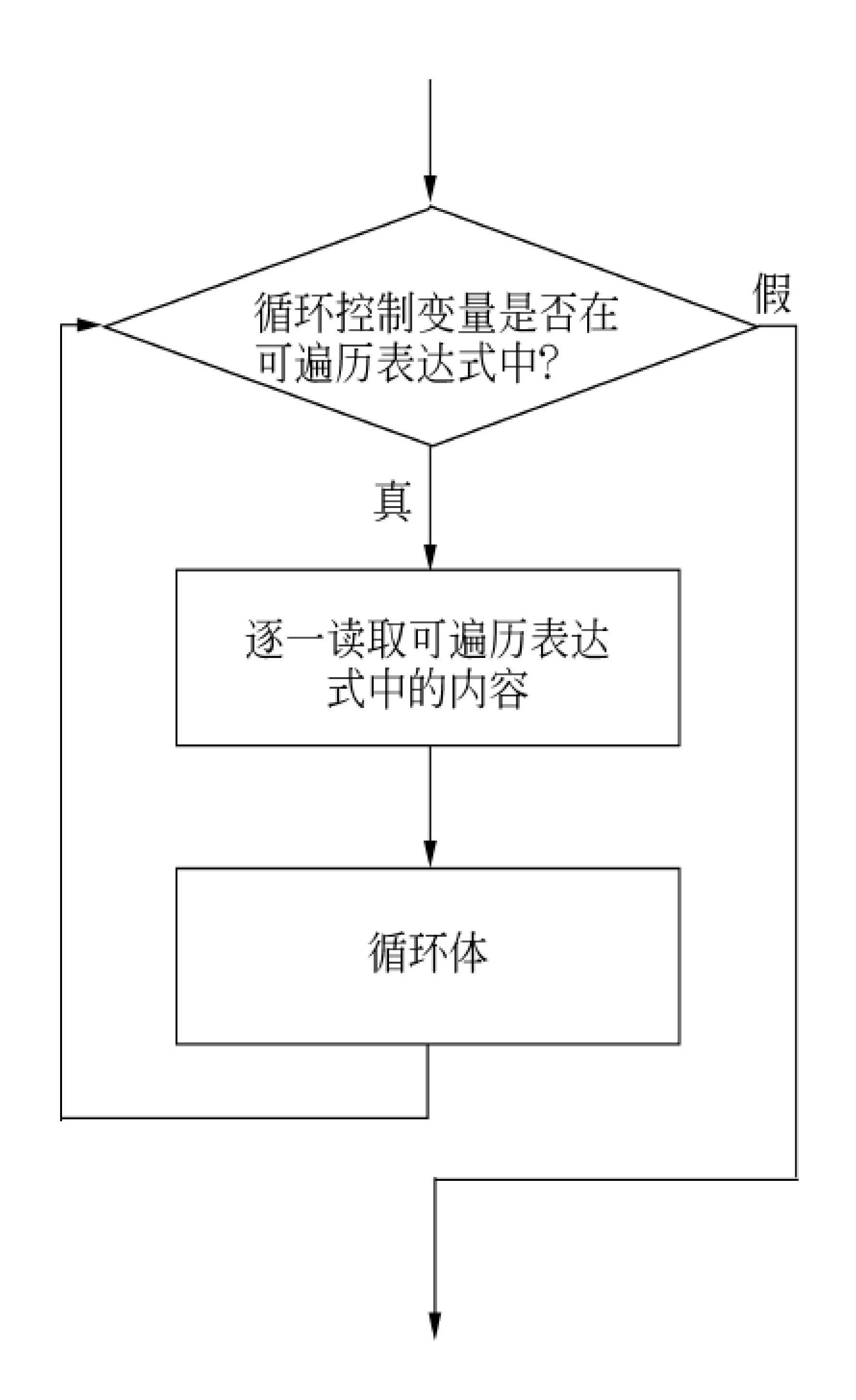


图 4-8　for 循环语句流程图

在 for 语句中,经常用 range()函数来控制循环次数和循环过程。range()函数的格式为:

```
range(start, stop, [step])
```

或

```
range(stop)
```



range()函数返回一个具有[start, start+step, start+2 * step, …,stop)结构的等差整数序列。range()函数有三个参数：①起始位置(start)表示数值序列的,默认为 0；②结束位置(stop)表示数值序列的结束值；③步长(step)表示等差序列中相邻两个数字的差值,默认为 1。

使用 range()函数时,需要注意它的一些特性：

(1) range()函数返回的是一个左闭右开[start,stop)的等差序列,即它不能取到结束位置(stop)的值,它只能取到“结束位置−步长”的值。例如,range(1,3,1)返回值为[1,2]。

(2) step 参数必须是非零整数,否则抛出 ValueError 异常。但是,它可以是正整数,也可以是负整数。

(3) range()函数的参数设置十分灵活,如果只给出一个参数,range(stop),则该参数表示结束位置,此时起始位置为默认值 0,步长为默认值 1。例如,range(3)返回值为[0,1,2]。如果给出两个参数,range(start, stop),则这两个参数分别表示起始位置和结束位置,此时步长为默认值 1。例如,range(1,3)返回值为[1,2]。

【例 4-12】 计算给定序列中所有奇数之和。

解答：本例中,对于给定的序列,其长度是已知的,因此,循环次数是确定的,可以使用 for 语句来实现。首先,通过列表定义一个已知序列。然后,用 for 语句遍历该列表,求出其中所有奇数的和。再次强调,for 语句中循环控制变量的值是自动更新的,切记不要画蛇添足,再额外处理。

```
numbers=[2,45,67,44,90,34,22,67,989,100,28,16,35]
odd_sum=0
for i in numbers:#依次读取 numbers 列表中的值
    if i %2 !=0:#如果值是奇数
        odd_sum+=n
print("数列中奇数之和为:",odd_sum)
```

【例 4-13】 统计字符串中元音字母的个数。

解答：字符串也是一种容器类函数,常用在 for 语句中作为可遍历表达式。

```
my_string=input("请输入一个字符串:")
a=e=i=o=u=0 #连续赋值语句,为各元音字母计数器赋初值
for char in my_string: #循环遍历输入的字符串,分别统计各元音字母个数
    if char=='a' or char=='A':#考虑元音字母有大小写的情况
        a+=1
    elif char=='e' or  char=='E':
        e+=1
    elif char=='i' or  char=='I':
        i+=1
    elif char=='o' or  char=='O':
        o+=1
    elif char=='u' or  char=='U':
        u+=1
    else:
        continue #强制跳到下一轮循环
print("单词中 a 或 A 的个数是:",a)
print("单词中 e 或 E 的个数是:",e)
print("单词中 i 或 I 的个数是:",i)
print("单词中 o 或 O 的个数是:",o)
print("单词中 u 或 U 的个数是:",u)
```

首先,输入一个字符串。然后,用 for 语句逐一判断该字符串中的各个字符是否属于元音字符,如果是,则元音字母计数器累加 1;如果都不是,则执行 continue 语句跳到下一轮循环。continue 语句和前面讲过的 break 语句都是中断循环体执行的方法,continue 语句中断循环体执行就后跳转到下一轮循环,而 break 语句直接结束循环。

为了实现复杂的算法,常常需要使用循环嵌套,即一个循环语句的循环体又是另一个或多个循环语句。例如,[[11,12,13],[21,22,23],[31,32,33]]是一个二维列表,它用于表示一个 3×3 的矩阵。这个二维列表的每一个元素又是一个列表,如果要读取每一个元素,则需要先遍历整个二维列表,然后再遍历该二维列表成员中的每一个元素。循环嵌套实现的难点在于确定各层循环中循环控制变量的含义和取值范围。



【例 4-14】 计算二维列表中奇数的个数。

解答：二维列表可以用来表示矩阵。这里，使用两层循环嵌套来读取矩阵中的每个元素。其中，第一层循环读取整行元素，第二层循环读取每行中的每个元素。

具体代码如下：

```
list1=[[11,12,13],[21,22,23],[31,32,33]]
n=0
for i in list1: #分别读取二维列表中的各元素，即 i 表示列表，如[11,12,13]
    for j in i: #分别读取 i 中的各元素，即 j 表示一个数，如 11
        if j%2 !=0:
            n=n+1
print("矩阵中有",n,"个奇数")
```

【例 4-15】 打印由任意字符组成的任意行数的等边三角形。

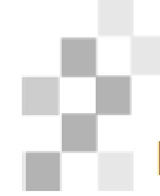
解答：本实例是根据用户输入的行数和字符构成等边三角形。编程前，需要认真分析清楚等边三角形的行数、每行需要绘制的空格数、字符数以及执行的循环次数之间的数学关系。

具体代码如下：

```
#打印字符组成的等边三角形
n=int(input("输入等边三角形的行数:"))
char=input("输入组成等边三角形的字符:")
for i in range(n):
    for j in range(n-i-1):
        print(" ",end='')
    for k in range(2*i+1):
        print(char,end='')
    print("\n")
```

说明：print(char,end='')语句表示输出字符 char 时不换行。

程序运行结果如图 4-9 所示。



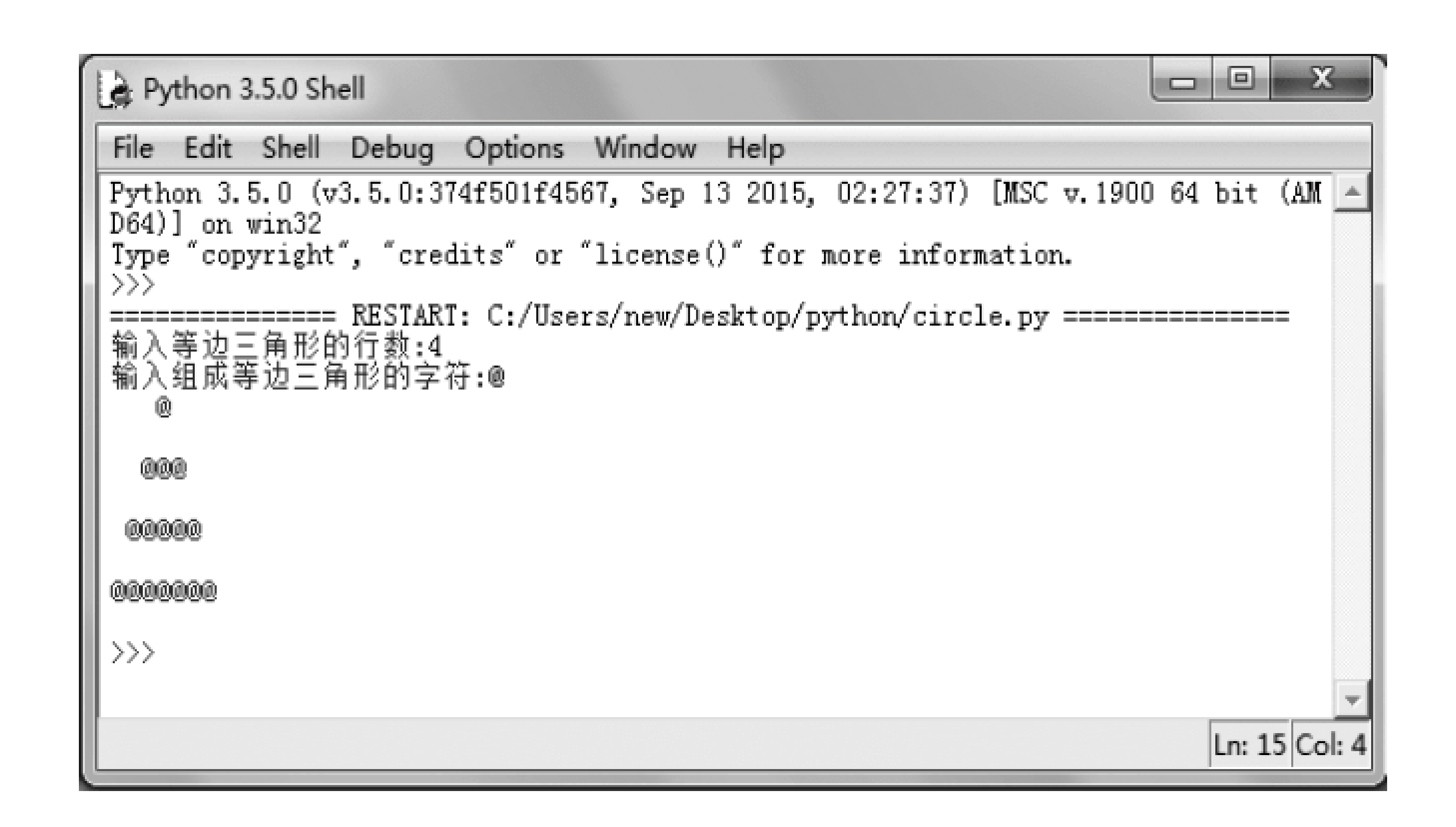


图 4-9　例 4-14 绘制等边三角形

4.2.4　循环跳转语句

在执行 while 语句或 for 语句等循环语句时，可能会出现提前结束本层循环，即强制退出当前循环体的执行的情况。Python 提供了两种强制退出循环的方法：break 和 continue。

1．break

break 语句用来跳出整个循环，即使 while 语句中条件表达式的值仍满足可执行条件或者 for 语句中可遍历表达式还没有被完全遍历完，循环语句都会被停止执行。当使用循环嵌套时，break 语句将跳出最深层的循环，继续执行后面的语句，而非停止执行所有的循环嵌套语句。

2．continue

continue 语句仅用来跳出本次循环，即停止执行当前循环体中的剩余语句，然后继续执行下一轮循环。

习　　题

1．检验输入的用户名、密码、电话号码等用户注册信息。要求：密码长度为 6～20；电话号码必须是数字，长度必须是 11 位。



提示：①判断输入的字符串是否只由数字组成可用 isdigit()函数；②获取字符串长度可用 len()函数。

2. 输入任意三个数，按从大到小排列输出。

3. 输出 2000—2016 年中的所有闰年。

提示：能被 4 整除却不能被 100 整除或者能被 400 整除的年份是闰年。

4. 计算 1～10000 所有的素数之和。

5. 输入一行英文，分别统计其中的元音字母个数。

6. 设计密码转换程序，要求：输入一行英文字符串，给出相应的加密字符串。加密规则是每个字母用其后的第 n 个字母代替(n 的值自行设计)。

7. 输出所有“水仙花数”。所谓“水仙花数”，是指一个三位数等于其各位数字的立方和，如 $153=1^3+5^3+3^3$。

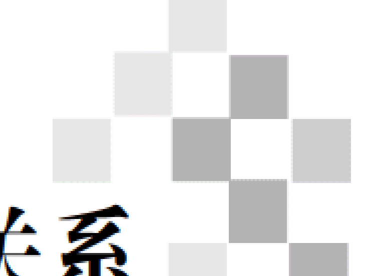
8. 口算练习程序。要求：随机产生两个 1～100 的整数，用户输入口算的结果(加、减、乘、除)，判断是否正确并给出相应提示信息。用户输入 000，则程序结束。随机数的产生可参考例 4-8 中的 randint 函数用法。

9. 输入年月日，输出这个日期是星期几。

提示：①1900.1.1 是星期一；②输入的日期在 1900.1.1～2100.12.31 内；③输入日期不合法时提示错误并重新输入；④输入 0 时程序结束。

10. 输入一个字符串，分别统计其中数字、字母和标点符号的个数。

第 5 章　计算机表示现实事物间关系



在第 2 章中提到过采用降维思维不仅可以表示现实世界中的一切事物，还能描述事物之间的各种关系。根据事物间相互作用和相互影响的状态，可将它们分为集合、线性、树形（层次）和网状 4 种关系类型。由于人脑具有多维多元的存储特点，它能实现各种关系的直接映射和整体表示，不需要经过转换或切分，并且能一次性地获取事物间的各种关系。但是，计算机存储系统仅是一个一维一元的线性空间，它只适合直接表示集合和线性关系。对于树形和网状等复杂数据关系，必须利用计算思维中的降维思维，对其进行转换和划分，将整体的一对多或多对多的非线性关系降维成局部的、一对一的线性关系。数据间的复杂关系只能由与其相邻的前驱对象和后继对象间接地、局部地表示。要想得到完整的数据关系，只能通过分步解析实现。

现实世界中存在着形形色色的事物，这些事物之间并不是孤立无依的，它们有着千丝万缕的联系。我们用计算机解决实际问题，不仅要能表示问题中包含的各类事物，还要能描述这些事物之间的复杂关系，从而完成将现实世界的实际问题映射到计算机世界的抽象模型的计算思维过程。

现实世界中各类事物间仅存在集合、线性、树形和网状 4 种关系，其拓扑结构图如图 5-1 所示。任何复杂关系都是这 4 种关系的组合。

1. 集合关系

具有某种共同特性的事物聚集在一起就构成一个集合。集合中的各个事物除属于同一个集合外无特别的关系，可将其视为是一个个独立个体。它们之间是平等、无序、不



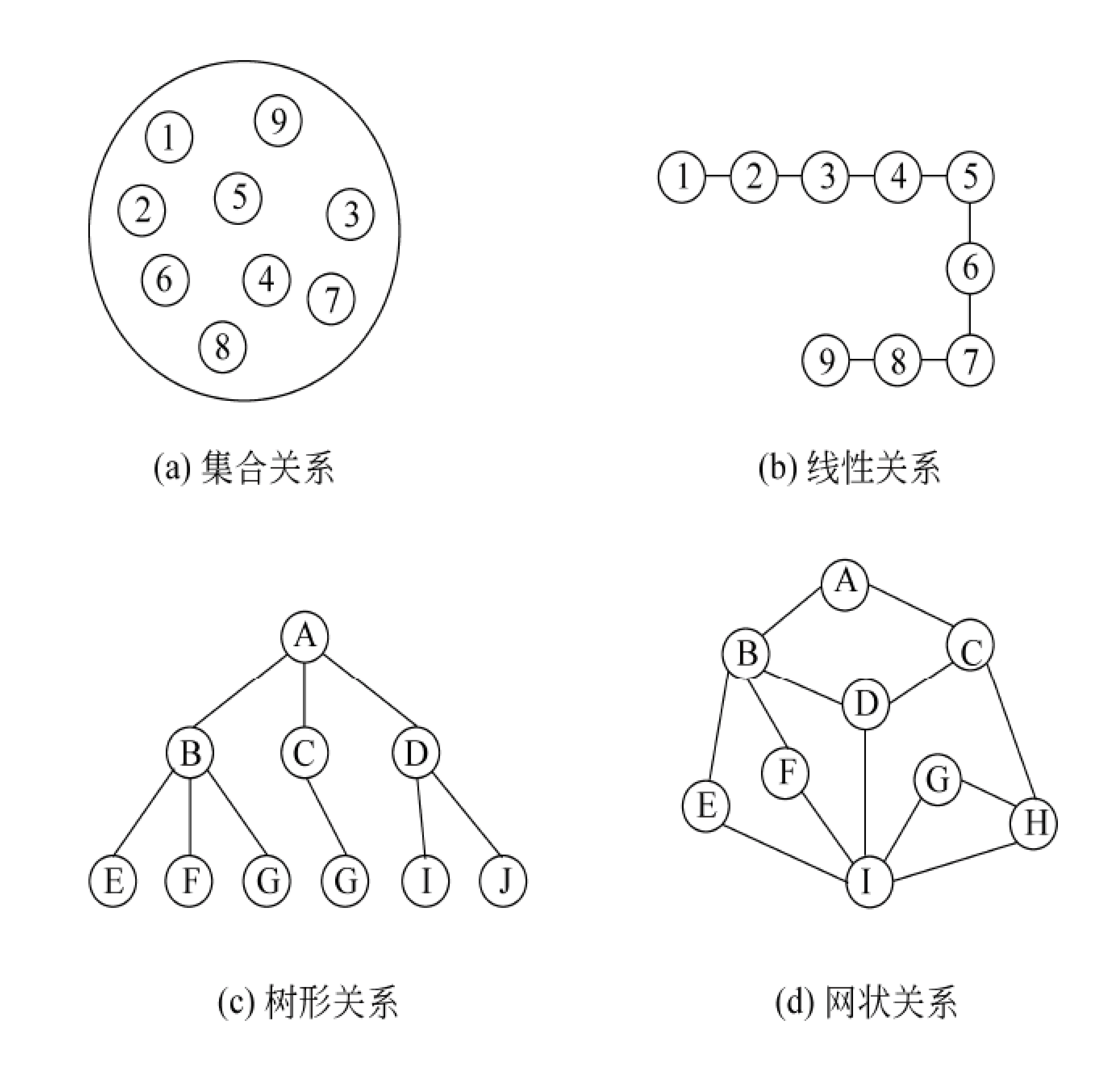


图 5-1　现实世界中事物间的逻辑关系

相关的，如学生集合、职员集合、用户集合等。

2. 线性关系

线性关系中各个事物之间是一对一的关系，它们构成一个有序集合，如超市中排队付款的顾客、连续剧中按序播放的剧集、书中顺序编码的书页等。

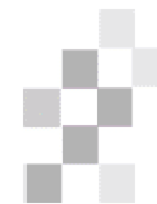
3. 树形关系

树形关系中各个事物之间是一对多的层次关系，它们之间存在“上下级”的不平等关系，如单位的组织机构关系、族谱中的家族关系、计算机中级联文件夹间的关系等。

4. 网状关系

网状关系中各个事物之间是多对多的复杂关系，即两两事物之间都有可能存在联系，如由站点构成的地铁交通图、学生与课程的关系等。

另外，对于相同的事物，由于解决的问题不同，它们之间的逻辑关系也不一定相同。例如，在课程管理系统中，学生之间是集合关系，他们在使用上没有差异，都是执行选课、查分



等操作;而在班级管理系统中,学生之间是树形关系,他们由于级别不同而具有不同的权限,班长权限最高——他可以查看和修改学生信息,普通学生具有一般权限——他仅能查看本人信息。由此可见,在分析事物间逻辑关系时,一定要联系实际问题,切忌循规蹈矩。

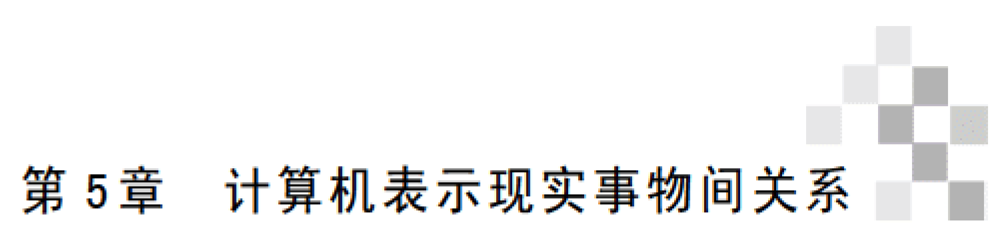
那么,这些现实世界中事物之间的关系如何用计算机表示呢? 物理上,计算机的内存是一个一维空间,因此存储时,我们只能将集合关系、线性关系、树形关系和网状关系转换为计算机能理解的一维结构。

进一步,如何在 Python 中定义一维结构呢? 在第 2 章中,我们学习了几种基本的数据类型,但是这些数据类型都只能用来表示单个数据,无法表达数据之间的复杂关系。这里,Python 提供了列表、元组和字典等能够同时表达多个数据且各数据可以具有不同数据类型的复合数据类型,如表 5-1 所示。其中,列表是使用中括号“[]”定义的一组有序数据集合,不同元素间用逗号“,”隔开,例如 a=['a','b','c']。列表定义完后可以被修改。元组是使用小括号“()”定义的一组固定数据集合,例如 a=('a','b','c')。元组的特性和列表的特性基本一致,但它定义后就不能被修改,因此所有具有改变性的操作都不能应用到元组中,例如 append()、extend()、insert()、remove()、pop()、sort()、reverse()等。字典是使用大括号“{}”定义的一组无序键-值对集合。字典和列表、元组的最大区别是,它的每个元素都是由键和值两部分组成,且键不能重复。和现实生活中的字典功能类似,字典表示的是键和值之间的映射关系,我们可以通过键来读取它所对应的值,例如 a={"Lily":23,"Kate",32,"John":18},可以用来表示人名及其年龄,其中,人名是键,年龄是值。字典定义后也可以被修改。字典中的键是任意不可变的数据类型,常用字符串或元组表示。

表 5-1 Python 中列表、元组和字典操作

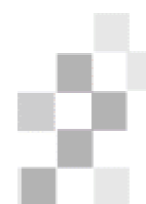
(其中,a=[1,2,3,4],t=(3,4,5,6),d={'a': 1, 'b': 2, 'c': 3})

数据类型	操作	定　义	说　明	实　例
列表	定义列表	List=[e1,e2,…]	用中括号[]定义列表	a=[1,2,3,4],结果[1,2,3,4]
	读取元素	List[index]	通过元素在列表 List 中的下标(index)读取该元素	a[0],输出 1



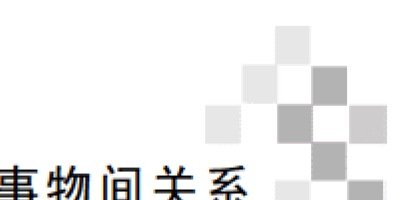
续表

数据类型	操作	定　义	说　明	实　例
列表	添加元素	List. append(e)	将元素 e 添加到列表 List 的尾部	a. append(5)，结果[1,2,3,4,5]
	插入元素	List. insert (index, data)	将元素 data 插入到列表 List 下标 index 位置	a. insert(3,6)，结果[1,2,3,6,4]
	插入列表	List. extend(List1)	将列表 List1 中各元素依次插入到原列表 List 的尾部	b=[5,6,7] a. extend(b)，结果[1,2,3,4,5,6,7]
	删除末尾元素	List. pop()	删除列表 List 尾部的元素，返回被删除的元素	a. pop()，输出 4
	删除指定元素	List. remove(e)	删除列表 List 中第一个值为 e 的元素，如果没有该元素则报错	a. remove(3)，结果[1,2,4]
	修改元素	List[index]=data	将列表 List 下标 index 所在元素重新赋值为 data	a[2]=6，结果[1,2,6,4]
	查找元素	e in List	判断元素 e 是否在列表 List 中。如果 e 存在，则返回 True；否则，则返回 False	1 in a，输出 True
	查找元素下标	List. index(e)	查找元素 e 在列表 List 中的下标位置，如果没有找到则报错	a. index(3)，输出 2
	计算元素次数	List. count(e)	计算元素 e 在列表 List 中出现的次数，如果没有出现则返回 0	a. count(1)，输出 1
	分片	NewList=List[index1: index2]	从列表 List 中同时读取从下标 index1 到下标 index2－1 的多个元素，返回一个新的列表	a[1: 3]，输出[2,3]
	连接两个列表	List1＋List2	将列表 List2 中各元素依次插入到列表 List1 尾部，并返回一个新的列表	b=[5,6,7] a＋b，输出[1,2,3,4,5,6,7]
	列表长度	len(List)	获取列表 List 的长度，即列表中元素个数	len(a)，输出 4
	排序	List. sort()	根据元素的数据类型，对列表中各元素进行比较和排序。一般是对具有同类元素的列表进行排序。该函数改变原列表，且无返回值	c=[1,2,1,4] c. sort()，输出[1,1,2,4] c=[1,'a',2,1] c. sort()，输出报错



续表

数据类型	操作	定　义	说　明	实　例
列表	反转	List. reverse()	按列表中元素的下标位置逆序输出各元素，对元素数据类型没有要求	c=[1,2,1,4] c. reverse()，输出[4,1,2,1] a=[1,'a',2,1] a. reverse()，输出[1,2,'a',1]
元组	定义元组	Tuple=(t1,t2,…)	用小括号“()”定义元组	t=(3,4,5,6)，结果(3,4,5,6)
	读取元素	Tuple[index]	通过元素在元组 Tuple 中的下标(index)读取该元素	t[0]，输出 3
	查找元素	t in Tuple	判断元素 e 是否在元组 Tuple 中。如果 e 存在，则返回 True；否则，返回 False	1 in t，输出 False
	查找元素下标	Tuple. index(e)	查找元素 e 在元组 Tuple 中的下标位置，如果没有找到则报错	t. index(4)，输出 1
	计算元素次数	Tuple. count(e)	计算元素 e 在元组 Tuple 中出现的次数，如果没有出现则返回 0	a. count(1)，输出 0
	分片	NewTuple=Tuple[index1：index2]	从元组 Tuple 中同时读取从下标 index1 到下标 index2－1 的多个元素，返回一个新的元组	t[1：3]，输出(4,5)
	连接两个元组	Tuple1＋Tuple2	将元组 Tuple2 中各元素依次插入到元组 Tuple1 尾部，并返回一个新的元组	f=(7,8) t＋f，输出(3,4,5,6,7,8)
	元组长度	len(Tuple)	获取元组 Tuple 的长度，即元组中元素个数	len(t)，输出 4
	元组转换为列表	list(Tuple)	将元组转化为列表	list(t)，输出[3,4,5,6]
	列表转换为元组	tuple(List)	将列表转化为元组	tuple(a)，输出(1,2,3,4)
字典	定义字典	Dic＝{key1：value1，key2：value2，…}	用大括号“{}”定义字典，其中 key：value 是一对键-值对，key 是键，value 是值。字典在存储时会根据元素的 hashcode 进行排序，可能会导致实际存储顺序和书写顺序不同	d={'a'：1，'b'：2，'c'：3}，结果{'c'：3，'b'：2，'a'：1}。

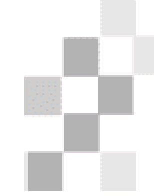


续表

数据类型	操作	定　　义	说　　明	实　　例
字典	读取元素	Dic. get(key)	通过键读取该键在字典 Dic 中对应的值。如果 key 存在,则返回 key 对应的值;否则,返回 None	d. get('c'),输出 3
	读取元素	Dic[key]	通过键读取该键在字典 Dic 中对应的值。如果 key 存在,则返回 key 对应的值;否则报错	d['c'],输出 3
	添加(修改)元素	Dic[key]=value	将键-值对(key, value)添加到字典 Dic 中。如果 key 存在,则将该键对应的原值修改为新值 value,并返回该新值;否则,向 Dic 中添加一个新的键-值对,并返回值 value	d['d']=4,输出 4
	删除指定元素	Dic. pop(key)	删除字典 Dic 中 key 所在的键-值对,并返回被删除值对中的值	d. pop('b'),输出 2
	查找元素	key in Dic	判断键 key 是否在字典 Dic 中。如果在 Dic 中,则返回 True;否则,返回 False	'b' in d,输出 True
	清除字典	Dic. clear()	清除字典 Dic 中的所有元素	d. clear(),结果{}
	复制字典	NewDic=Dic. copy()	将字典 Dic 中的各元素复制给一个新的字典 NewDic	k=d. copy(),结果{'a': 1, 'b': 2, 'c': 3}
	更新字典	Dic. update(Dic1)	用字典 Dic1 中的各键-值对更新 Dic 中的各键-值对。如果 Dic1 中的键在 Dic 中存在,则用该键在 Dic1 中对应的值更新 Dic 中对应的值;否则,将该键-值对添加到 Dic 中	m={'c': 9, 'f',7} d. updata(m),结果{'c': 9, 'b': 2, 'a': 1, 'f': 7}
	读取所有的键值	Dic. keys()	以列表返回字典所有的键	d. keys(),输出 dict_keys(['b', 'c', 'a'])
	读取所有的值	Dic. values()	以列表返回字典所有的值	d. values(),输出 dict_values([2, 3, 1])

【例 5-1】 判断下面列表操作是否正确,并解释错误原因。设列表 a=[1,2,3]。

(1) 在列表尾部增加一个新元素 5,执行 a[4]=5。



(2) 同时获取列表中 2 和 3 这两个元素,执行 a[1:2]。

(3) 将一个列表 b=[4,5,6]中的各元素插入到原列表尾部,执行 a.append([4,5,6])。

(4) 在列表末尾同时添加"4,5,6"等多个元素,执行 a.append(4,5,6)。

解答:(1)错误。列表虽然是可以被修改的,但是下标 4 在列表中并不存在,即它在内存中没有被分配空间。我们不能给一个不存在的内存空间赋值。因此,只能通过 a.append(5)方法开辟一个新的内存空间,并对该空间赋值。

(2) 错误。对列表执行分片操作时(List[index1:index2]),第二个下标位置上的元素是取不到的,仅能取到该下标-1 位置上的元素。要想同时读取 2 和 3,只能执行 a[2:4]操作。

(3) 错误。执行 List.append(e)操作时,e 可以是任意类型,但仅能被视为是一个元素添加到列表尾部。因此,a.append([4,5,6])的执行结果是[1,2,3,[4,5,6]],而非[1,2,3,4,5,6]。

(4) 错误。List.append(e)一次仅能添加一个元素。这里可以通过 a.extend([4,5,6])实现所需操作。

【例 5-2】 判断下列元组操作是否正确,并解释错误原因。设元组 t=(3,4,5,6)。

(1) 向元组末尾添加一个新元素 7,执行 t.append(7)。

(2) 删除元组中一个元素 4,执行 t.remove(4)。

(3) 在元组中下标 3 的位置插入一个元素 8,执行 t.insert(3,8)。

(4) 将元组中元素 5 的值修改为 1,执行 t[2]=8。

解答:元组是一种固定的数据类型,它一旦定义是不能再被修改的。因此,一般不对元组执行添加、删除、插入或修改等具有改变性的操作。在设计系统前,对于变量数据类型的确定,一定要三思而后行,以免在后续开发中出现预先定义的变量不能满足实际需求等问题。

针对(1)~(4)的错误,有两种解决方案:一是将元组数据类型转换为灵活性较高的列表数据类型,在列表中执行完添加、删除、插入或修改等操作后,再将列表数据类型转换回元组数据类型。二是对元组执行查找、分片、重组等操作,并以元组嵌套的方式定义



新元组。虽然得到的结果是一个二层嵌套元组，但它并不影响实际应用。

题目(1)的改正如下。

方法一：元组-列表-元组。

```
t=(3,4,5,6)
ListTemp=list(t)
ListTemp.append(7)
t=tuple(ListTemp)
```

结果：(3,4,5,6,7)。

方法二：元组嵌套。

```
t=(3,4,5,6)
(t,7)
```

结果：((3,4,5,6),7)。

题目(2)的改正如下。

方法一：元组-列表-元组。

```
t=(3,4,5,6)
ListTemp=list(t)
ListTemp.remove(4)
t=tuple(ListTemp)
```

结果：(3,5,6,7)。

方法二：元组嵌套。

```
t=(3,4,5,6)
(t[0:t.index(4)], t[t.index(4)+1:len(t)])
```

结果：((3,), (5, 6))。

注意：(3,)是一个仅有一个元素的元组，而(3)则是一个表达式。

题目(3)的改正如下。

方法一：元组-列表-元组。

```
t= (3,4,5,6)
ListTemp=list(t)
ListTemp[ListTemp.index(5)]=8
t=tuple(ListTemp)
```

结果：(3,4,8,6)。

方法二：元组嵌套

```
t= (3,4,5,6)
(t[0:3], 8, t[3:len(t)])
```

结果：((3，4，5)，8，(6,))。

题目(4)的改正如下。

方法一：元组-列表-元组。

```
t= (3,4,5,6)
ListTemp=list(t)
ListTemp[ListTemp.index(5)]=1
t=tuple(ListTemp)
```

结果：(3,4,1,6)。

方法二：元组嵌套。

```
t= (3,4,5,6)
(t[0:t.index(5)], 1, t[t.index(5)+1:len(t)])
```

结果：((3，4)，1，(6,))。

【例 5-3】 判断下列字典操作是否正确,并解释错误原因。设字典 d={'a': 1，'b': 2，'c': 3}。

(1) 读取字典中第一个位置的元素,执行 d[0]。

(2) 删除字典中最后一个位置的元素,执行 d.pop()。



(3) 将 d['d']=4 插入到字典中键值为'b'的位置后，执行 d.insert[3,'d']=4。

(4) 将字典中键为'a'的值修改为 1，执行 d['a']=5。

解答：(1) 错误。字典是用键 key 读取其映射的值的，在字典中没有下标的概念。所以，在字典中执行 d[0]操作，字典无法识别 0 的含义。读取字典中第一个位置的元素，应该执行 d['a']。

(2) 错误。原因和(1)相同，字典中涉及添加、读取或删除值等操作，都必须通过键 key 来完成。因此，删除最后一个位置的元素，应该执行 d.pop('c')。

(3) 错误。字典是一个无序数据类型，它输出的顺序是按照字典中各值的 id 进行排序的。因此，字典无法执行在某个指定位置插入一个键-值对的操作。只能通过执行 d.['d']=4 向字典中随机地插入一个键-值对，无法控制该键-值对的插入位置。

(4) 正确。字典是一个可修改的数据类型，可以通过指定键来修改它所映射的值。

5.1　集合关系

集合关系体现了事物间的聚集。在 Python 中，可以用列表或元组来直接创建数据间的集合关系。集合除了将各数据元素聚集在一起外，别无他用。集合常用来表示或处理一组数据，例如由学号组成的学生集合、由账号组成的用户集合等。我们经常会对集合中的元素执行查找、添加、删除、修改和排序等操作。

【例 5-4】　设计学生信息管理系统。

解答：这是一种开放式命题，题目完成的质量主要取决于设计者对需求的把握程度。任何实际问题都要经过适度地抽象和化简，才能转换为计算机所能求解的数学模型。在简化过程中，“恰当”十分重要，简化后的模型既不能过易，也不能太难，抓住本质最佳。同时，在合理和简化之间要做出折中。

针对本题，经过调研和已有经验，可以确定学生信息管理系统主要包括对学生学号的查找、添加、删除、修改和排序等操作。这里采用列表创建用于存储学生学号的集合。为了增强程序的健壮性，在对列表中的元素执行添加、删除或修改等操作前，需要判断该

元素是否存在。

```
stuList=[]#创建一个空列表

#添加学生
stuIns=input("Input a new student ID:\n")
if stuIns in stuList: #判断需要添加学生的学号是否存在
    print("This student already exists!")
else:
    stuList.append(stuIns)
    print("The current students are:", stuList)

#删除学生
stuDel=input("Input the student ID you want to remove:\n")
if stuDel in stuList: #判断需要删除学生的学号是否存在
    stuList.remove(stuIns)
    print('The current students are:', stuList)
else:
    print("This student does not exist!")

#查找学生
stuFind=input("Input the student ID you want to find:\n")
if stuFind in stuList:#判断需要查找学生的学号是否存在
    print("This student is found!")
else:
    print("This student is not found!")

#修改学生
stuRep=input("Input the old student ID you want to replace:\n")
if stuRep in stuList: #判断需要修改学生的学号是否存在
    stuNew=input("Input the new student ID:\n")
    oldIndex=stuList.index(stuRep)
```

```
        stuList[oldIndex]=stuNew
        print("The current students are:", stuList)
    else:
        print('The student you want to replace does not exist!')
```

5.2　线性关系

线性关系主要是表达事物之间一对一的有序关系。表示线性关系的数据类型称为线性表,其定义为:线性表(Linear List)是零个或多个元素的有序序列。

在线性表中,各元素按某种属性进行排序,像“线”一样首尾相接,具有确定的相对位置,如图 5-2 所示。线性关系中的元素可以细分为头元素、尾元素和中间元素三种类型。其中,头元素只有后继元素,尾元素只有前驱元素,而中间元素则同时具有前驱元素和后继元素。

在计算机中,具有线性关系的数据类型统称为线性表。Python 可以利用列表或元组中下标间的有序关系来直接创建线性表。在实际问题中,我们经常会对具有线性关系的数据元素执行查找、添加、删除、修改和排序等操作。但是,线性关系具有有序特性,所以实现上述操作的难度比集合要大。

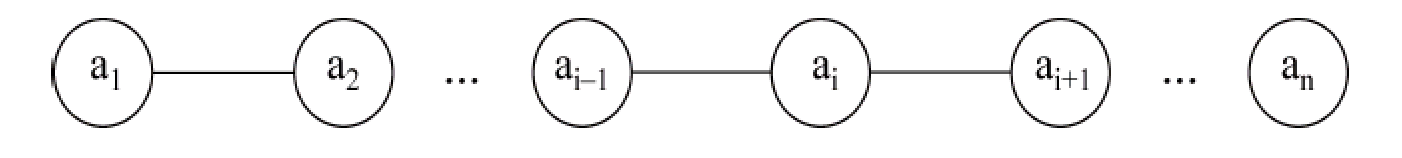


图 5-2　线性关系

实际上,线性关系中的有序关系可以细分为三种关系类型:普通线性关系、队列关系和栈关系。其中,普通线性关系就是一对一的有序关系;队列线性关系是具有“先进先出”约束的线性关系;栈关系是具有“后进先出”约束的线性关系。

1. 普通线性关系

普通线性关系是最简单、最直接的一种线性关系。其他两种特殊的线性关系,如队列关系和栈关系,都是在普通线性关系基础上演变而来的。它继承了线性表中的所有操

作,具有广泛的应用。在现实生活中,具有一对一有序关系的各个事物之间均可以用普通线性关系表示。例如,一年中的各个月份、首尾相连的各节车厢、磁带中存储的各首歌曲等。

【例 5-5】 设计学生信息管理系统,要求学生按照学号排序。

解答: 题目要求按学号顺序(通常从小到大)存储学生信息,因此,可以使用列表表示学生之间的线性关系。这里需要注意,对学生列表执行添加或删除等操作时,需要逐个移动列表中其他元素的位置,以时刻保证线性列表的有序特性。值得庆幸的是,Python 提供的列表添加或删除元素的内置函数,如 List.insert(index, data)和 List.remove(e)等(详见表 5-1),均具有自动移位功能,无须编程者额外操作。另外,学号在形式上一般用数字表示,但实际应用时一般定义为字符串类型。

具体代码如下:

```
stuList=[]#创建一个空列表

#添加学生
stuIns=input("Input a new student ID:\n")
if stuIns in stuList: #判断需要添加学生的学号是否存在
    print("This student already exists!")
else:#找到添加学生的合适位置
    if len(stuList)==0:
        stuList.insert(0,stuIns)
    else:
        for i in range(len(stuList)):
            if stuList[i] >stuIns:
                stuList.insert(i,stuIns)
                break
            print("The current students are:", stuList)
            print("The current number of students is:",len(stuList))

#删除学生
```



```
stuDel=input("Input the student ID you want to remove:\n")
if stuDel in stuList: #判断需要删除学生的学号是否存在
    stuList.remove(stuDel)
    print("The current students are:", stuList)
    print("The current number of students is:",len(stuList))
else:
    print("This student does not exist!")

#查找学生
stuFind=input("Input the student ID you want to find:\n")
if stuFind in stuList: #判断需要查找学生的学号是否存在
    print('This student is found!')
else:
    print('This student is not found!')

#修改学生
stuRep=input("Input the old student ID you want to replace:\n")
if stuRep in stuList: #判断需要修改学生的学号是否存在
    stuNew=input("Input the new student ID:\n")
    oldIndex=stuList.index(stuRep)
    stuList[oldIndex]=stuNew
    stuList.sort()
    print("The current students are:", stuList)
else:
    print("The student you want to replace does not exist!")
```

说明：例 5-5 代码中加粗部分是和例 5-4 代码的主要区别。

思考：(1) 如果学生信息管理系统中不仅需要存储学生的学号，还需要存储学生的总成绩，应该如何修改上述程序？

(2) 如果不直接利用 Python 提供的对列表添加或删除元素的内置函数，如何实现对有序列表中元素的添加或删除操作？

2. 队列关系

队列关系是一种特殊的线性关系，它只允许元素在线性表的头部(称为队头)进行删除(读出)，在线性表的尾部(称为队尾)进行插入(读入)，如图 5-3 所示。队列具有"先进先出"的特性，这和现实世界中队列的性质相同。当排队时，总希望大家遵守规则，不要插队。因此，在使用有限资源时，当需求方使用速度和服务方响应速度产生不匹配时，队列提供了一种合理的公平对待原则。队列在实际生活中应用十分广泛，例如，银行业务办理、超市结账、打印机的缓冲区、操作系统的作业队列等均可以使用队列实现。

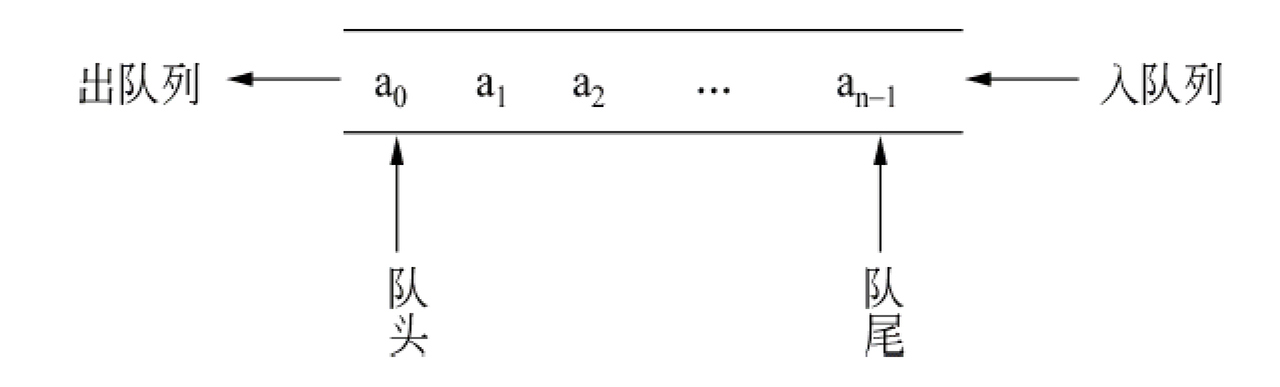


图 5-3　队列示意图

【例 5-6】 运动会比赛日程安排：某运动会设立 N 个比赛项目，每个运动员最多参加 M 个比赛项目，如何安排比赛才能使比赛组数最少？

解答：本题目需要解决"如何安排一个合理比赛日程"的问题，它要求既不能让同一运动员参加的比赛项目在同一单位时间进行，又要使总的竞赛日程最短，即能够在同一时间内比赛的项目数最多。通过分析，本题目可以抽象为一个划分"无冲突子集"的数学模型。

说明："无冲突子集"的数学描述为：已知集合 $A=\{a_1, a_2, \cdots, a_n\}$($n\in N$，当 $i\neq j$ 时，$a_i\neq a_j$)，将 A 划分为 K 个互不相交的子集 A_1，A_2，…，A_k($k\leqslant n$，当 $i\neq j$ 时，$A_i\cap A_j=\varnothing$)。

解决"无冲突子集"问题的方法如下所示：

初始化一个存储所有比赛项目的队列 queueItem
初始化一个记录比赛项目间互相冲突的二维矩阵表 clashMatrix，0 表示不冲突，1 表示冲突
定义记录组号的变量 groupNum，初始化为 0



```
定义一个列表存储每组中的项目 groupItem,初始化为[0]
#通过循环方式依次读取各比赛项目
①从 queueItem 队列的队头依次取出各比赛项目 i:
根据 clashMatrix 中的冲突记录,循环判断比赛项目 i 是否和 groupItem 中记录的
本组其他比赛项目冲突:
    如果不冲突,则:
            将该比赛项目 i 添加到组号为 groupNum 的列表 groupItem 中
            将该比赛项目 i 从队列 queueItem 中删除
    如果冲突,则:
            将该比赛项目 i 插入到队列 queueItem 尾部
    如果 queueItem 队列不为空,则:
        groupNum=groupNum+1
        跳转到①继续执行
    如果 queueItem 队列为空,则:
        程序结束
```

为了更好地理解上述算法,下面举一个简单实例。例如,学校运动会设有 5 个比赛项目,I={0,1,2,3,4},有 3 名运动员报名参加的项目分别为 P_1=(0,1,4),P_2=(2,3),P_3=(1,3,4),由此可知,各项目之间的冲突关系 CR={(0,1),(0,4),(1,4),(2,3),(1,3),(3,4)}。为了获得最少比赛组数的算法的具体步骤如下:

(1) 初始组号为 0 且当前该组号包含零个比赛项目。从 I 中取出第一个比赛项目 0,由于组号 0 中没有比赛项目,即没有与比赛项目 0 相冲突的其他比赛项目,因此,直接将比赛项目 0 添加到组号为 0 的项目组中,并将 0 从 I 中删除。此时,I={1,2,3,4},group0={0}。

(2) 从 I 中取出第二个比赛项目 1,根据项目冲突关系 CR 表,得知比赛项目 1 和比赛项目 0 相冲突,因此,不能将比赛项目 1 添加到组号为 0 的项目组,将 1 插入到 I 的末尾。重复执行上述操作,直至 I 中所有元素均读取完毕。此时,I={1,3,4},group0={0,2}。

(3) 创建一个新的组号 1,重复执行步骤(1)和步骤(2),得 I={3,4},group1={1}。

(4) 创建一个新的组号 2,重复执行步骤(1)和步骤(2),得 I={4},group2={3}。

(5) 创建一个新的组号 3,重复执行步骤(1)和步骤(2),得 I=∅,group3={4}。

由此可知,最少需要 4 个组才能安排开各类比赛项目。

上述算法的程序代码如下:

```
numItem=5 #比赛项目的数量
numPlayer=3 #运动员的数量
queueItem=[0,1,2,3,4] #用队列存储所有比赛项目
playerItem=[[0,1,4],[2,3],[1,3,4]] #用二维列表分别存储各队员的比赛项目
clashMatrix=[] #用二维列表,即矩阵存储各比赛项目间的冲突
groupNum=0 #定义组号变量
groupItem=[[]] #用二维列表存储各组项目的列表

#根据 playerItem 构造冲突矩阵 clashMatrix
for i in range(numItem):
    temp=[]
    for j in range(numItem):
        temp.insert(j,0)
        for k in range(numPlayer):
            if i !=j and i in playerItem[k] and j in playerItem[k]:
                temp[j]=1
    clashMatrix.append(temp)

#构造存储各组比赛项目的二维列表 groupItem
preOut=-1 #存储队列中前一次输出的元素,开始时队列无前一次输出的元素,因此
初始化为-1
while queueItem: #判断队列是否为空
    outItem=queueItem[0] #从队头读取元素
    insertBool=True #定义是否插入队头元素的开关变量
    if preOut >=outItem: #判断是否开始一个新的比赛组,因为各比赛项目按编
号从小到大排序,所以如果前一个比赛项目编号比当前输出的比赛项目编号大,则
```



```
说明所有比赛项目已经遍历一遍。等号用于最后一次比较
        groupNum=groupNum+1
        groupItem.append([])

    for x in groupItem[groupNum]:#判断当前读取的比赛项目和将其插入到的比赛组中其他项目是否冲突
        if clashMatrix[x][outItem]==1:#如果冲突,则将该比赛项目从队列中删除并添加到队尾
            preOut=outItem
            queueItem.remove(outItem) #从队头出队
            queueItem.append(outItem) #从队尾入队
            insertBool=False
    if insertBool==True or not groupItem[groupNum]:#如果当前比赛组中没有任何比赛项目与队头元素相冲突,则将该队头元素添加到该比赛组中,并从原队列中删除
        preOut=outItem
        groupItem[groupNum].append(outItem)
        queueItem.remove(outItem) #从队头出队

print(groupItem)
```

说明：本题利用“队列中存储的各类比赛项目编号是有序排列”这一特性,得出如果前一次读取的比赛项目编号 preOut>当前读取的比赛项目编号 outItem,则队列中所有元素已经被遍历完一遍,需要再创建新的比赛组。另外,preOut==outItem,用于最后一次比较,即队列中仅有一个元素的情况。

通过本题可以看出,虽然问题的解决方法(算法)与实际的程序代码在形式上还存在一些差异,但这都只是在编程过程中的一些经验和技巧,以及编程语言特定的语法规则。解题的核心和关键还是算法,算法是编写程序的根基,没有算法,何谈程序。

思考：如何将上述题目设计得更加灵活？例如各类比赛项目、比赛项目数量、运动员数量、每个运动员的参赛项目等信息均由用户输入。这里需要注意,我们是采用队列来

存储各类比赛项目的,因此,只能从队头添加元素。

【例 5-7】 模拟客户到银行办理业务的过程,其主要描述为:

(1) 客户去取号机取号。

(2) 客户按取号顺序排队。

(3) 银行业务员按客户的取号顺序叫号。

(4) 过号未到或客户晚到者均直接作废,需重新取号。

模拟过程为:输入命令'i'或'I',表示有客户到达并需要取号;输入命令'o'或'O',表示业务员叫下一位客户办理业务;输入命令'y'或'Y',表示客户存在;输入命令'n'或'N'表示客户不在;输入命令'a'或'A',表示有客户到达但已取过号码;输入命令'q'或'Q',表示不再接收客户排队。

解答:这是一个典型的队列问题,业务员按照客户的取号顺序执行入队(添加一个客户到队尾)或出队(从队头叫走一个客户)操作。值得注意的是,过号或晚到者会将将其旧号码直接作废。

问题求解的伪代码如下所示:

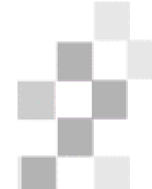
```
定义一个客户排队等候的队列 waitQueue
定义一个用户的取号变量 waitNum
定义一个业务员的叫号变量 callNum
循环判断用户输入的命令:
    如果输入命令=='i'或'I',则:
        对新来的用户取号并将其插入到队列尾部
    如果输入命令=='a'或'A',则判断已取号的用户其号码是否正确:
        如果该号码在当前等待队列中,则:
            如果恰好是当前队列头部元素,则:
                叫该用户办理业务
            否则:
                该用户继续等待
        否则:#即该号码不在当前等待队列中
            如果该号码小于当前等待队列的头部元素,则:
```



```
            该号码已过期,需重新取号并将其插入到尾部
        否则:
            输入的已取号码有误,需重新输入
如果输入命令=='o'或'O',则:
    如果当前等待队列不为空,则:
        业务员从当前队列头部叫一个客户
        如果该客户存在,则:
            对该客户办理业务
        否则:
            该客户过期作废
如果输入命令=='q'或'Q',则:
    程序退出
```

具体代码实现如下:

```
waitQueue=[] #客户排队等候队列
waitNum=0
callNum=0
while True:
    command=input("请输入一个命令:\n")
    if command=='i' or command=='I': #取号并插入到队头
        waitNum=waitNum+1
        waitQueue.insert(0,waitNum)
        print("请取号:", waitNum)
    if command=='a' or command=='A':
        inputNum=int(input("请输入你的取号码:"))
        if inputNum in waitQueue: #所输号码在当前等待队列中
            if inputNum==waitQueue[len(waitQueue)-1]: #所输号码正好是队
尾元素
                print("正好该你了", inputNum)
                waitQueue.pop()
            else:
                print("还没轮到你,请继续排队等候!")
```




```
        else: #所输号码不在当前等待队列中
            if inputNum <waitQueue[len(waitQueue)-1]: #该号码已过期作废
                waitNum=waitNum+1 #重新取号并插入到队头
                waitQueue.insert(0,waitNum)
                print("你的",inputNum,"已经过期,请重新取号:", waitNum)
            else:
                print("输入号码有误,请重新输入!")
                continue
    if command=='o' or command=='O': #叫号并从队尾离开
        if waitQueue:
            callNum=waitQueue[len(waitQueue)-1]
            print("请客户",callNum,"到柜台办理业务")
            ans=input("该客户是否在?是 or 不是")
            if ans=='y' or ans=='Y':
                print("该你了", callNum)
            if ans=='n' or ans=='N':
                print("该客户不在", callNum)
            waitQueue.pop()
        else:
            print("当前无等候的客户!")
    if command=='q' or command=='Q':
        print("程序结束!")
        break
```

3. 栈关系

栈关系是另外一种特殊的线性关系，它只允许元素在线性表的尾部(称为栈顶)进行插入和删除操作，在线性表的头部(称为栈底)不执行任何操作，如图 5-4 所示。栈具有"后进先出"的特性。从表面上看，栈违背了公平原则，但它却有特殊用途。例如，网页浏览中的"后退"功能、Word 软件中的"撤销"功能、电梯中乘客的进出顺序(先进电梯的乘客会因为位置靠里而后出电梯)等，都体现出时间上的"正序输入、逆序输出"特点。因

此，栈提供了一种有效解决“逆序”问题的方法。

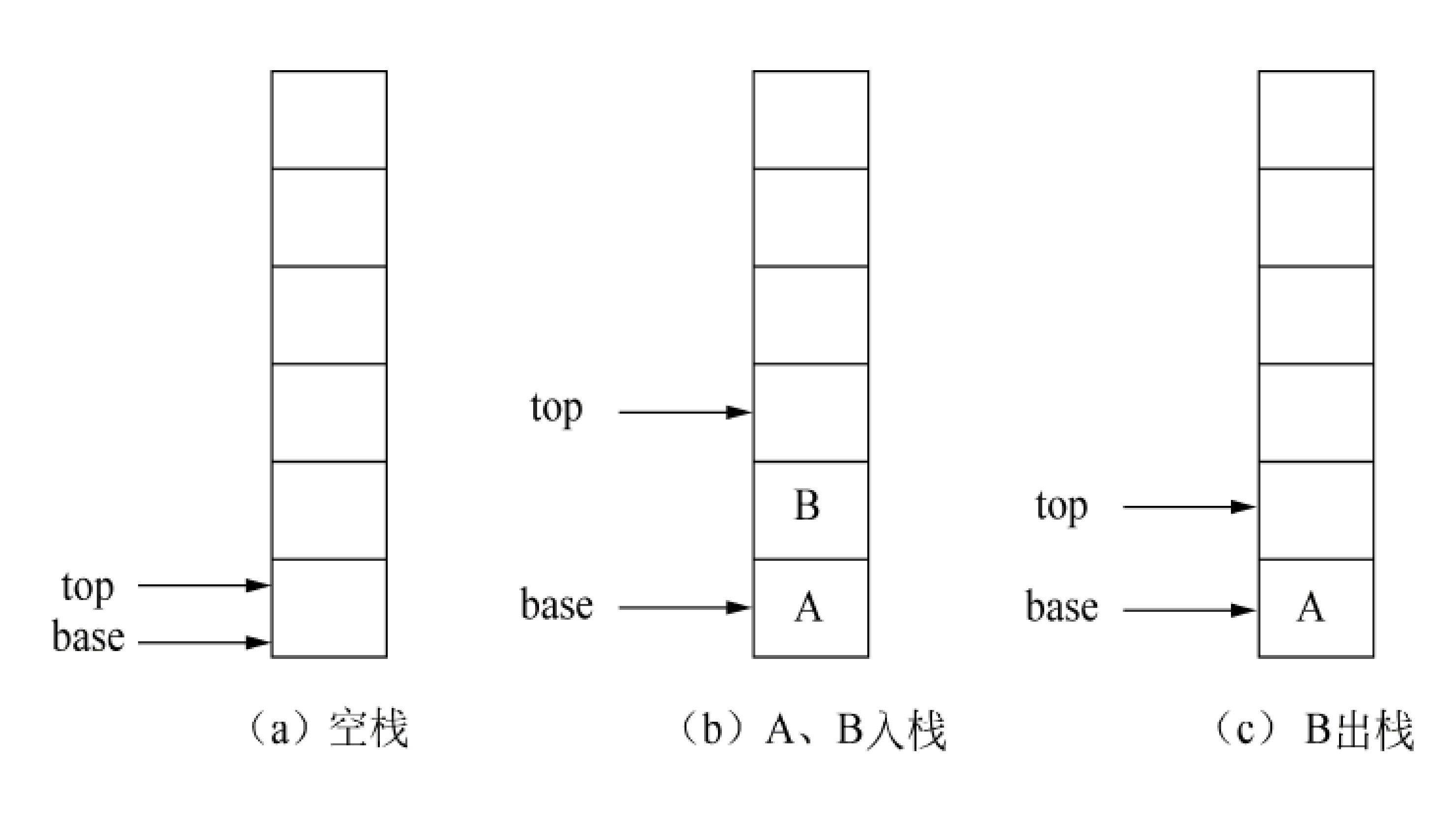


图 5-4　栈关系

【例 5-8】　正整数的数制转换，以十进制转二进制为例。

解答：十进制整数转换为二进制整数采用“除 2 取余，逆序排列”法，即需要对计算出的余数逆序输出。因此，可以使用栈存储各位余数。另外，题目中对原十进制数所做的“除 2 取余”操作是重复进行的，可用循环实现。

扩展：十进制整数转换为二进制整数的具体做法是：用 2 整除十进制整数，可以得到一个商和余数；再用 2 去除商，又会得到一个商和余数……如此进行下去，直到商为 0 时为止，然后把先得到的余数作为二进制数的低位有效位，后得到的余数作为二进制数的高位有效位，依次排列输出。例如，$(10)_{10}=(1010)_2$。

解决上述问题的伪代码形式如下所示：

```
输入一个十进制正整数 decNum
定义存储余数的栈 remStack
#循环实现十进制转二进制操作
①如果 decNum !=0,则:
rem=decNum %2
decNum=int(decNum/2)
将 rem 插入到 remStack 的栈顶,即入栈
跳转到①
依次读取 remStack 栈顶元素,即出栈
```

具体代码如下：

```
decNum=int((input("请输入一个正整数:\n")))
remStack=[] #定义一个空栈
while decNum !=0:
    rem=decNum %2 #取余数
    decNum=int(decNum/2) #取整数
    remStack.append(rem) #将余数插入到栈的顶部(队列的尾部),即入栈
for i in range(len(remStack)-1,-1,-1):
    print(remStack[i],end='') #从栈的顶部(队列的尾部)依次输出各位余数,且
不换行
```

说明：(1) range(初始值,结束值,步长)：range函数用于生成等差数列,它有三个参数。使用时,初始值和步长可以省略,其默认值分别为0和1,即等差数列从0开始,步长为1。步长可以是正整数,也可以是负整数。值得注意的是,range函数无法取到结束值。

(2) print(end=''),表示print输出时不换行。

【例5-9】 设计一个编译器,解决括号匹配问题(括号类型包括小括号、中括号和大括号)。

解答：括号匹配问题是指判断同种类型的左括号和右括号是否成对出现,如[([][])],{[(())}等。

在解决上述问题之前,首先需要了解计算机的特性。计算机在处理问题时和人有着本质的区别,它没有全局观,无法像人一样对事物进行整体输入、并行处理和统筹安排。它只能按照程序员设定的指令顺序输入数据、处理数据和输出结果,这就导致计算机只能"看见"当前时间点以前的数据,而无法预知未来将要发生什么。因此,当用户输入一个符号,如"["时,计算机并不知道后面是否有"]"与之相匹配,唯一的解决方法就是将无法判断匹配性的符号暂存起来,等后续出现与之相匹配的符号时,再进行判断。在判断符号匹配的过程中不难发现,被暂存的各个符号具有"逆序"特性,即后被存储的符号需要先进行判断。因此,我们采用栈解决这类问题。

解决上述问题的伪代码形式如下所示：



```
输入一组括号 pare
定义存储待匹配符号的栈 pareStack
依次读取 pare 中的括号 i:
    如果 i 是左括号,则:
        将 i 压入栈 pareStack 中
    如果 i 是右括号,则:
        如果栈为空,则:
            没有与该右括号匹配的左括号,pare 不匹配
        否则:
            如果 i 和 pareStack 当前栈顶的元素匹配,则:
                将 pareStack 的栈顶出栈
            否则:
                没有与该右括号匹配的左括号,pare 不匹配
如果 pareStack 为空,则:
    pare 匹配
否则:
    pare 不匹配
```

具体代码如下:

```
pare=input("请输入一组由小括号、中括号、大括号组成的括号串:\n")
pareStack=[]
state=1
for i in pare:
    if i=='(' or i=='[' or i=='{': #如果 i 是左括号
        pareStack.append(i)
    if i==')' or i==']' or i=='}':#如果 i 是右括号
        if not pareStack:#如果栈为空
            state=0
        else:
            bottom=pareStack[len(pareStack)-1]
            #如果 i 和 pareStack 当前栈顶的元素匹配
            if i==')' and bottom=='(' or i==']' and bottom=='[' or i=='}'
```

```
and bottom=='{':
                pareStack.pop() #将栈顶元素出栈
            else:
                state=0
if not pareStack and state:
    print("输入的括号串匹配!")
else:
    print("输入的括号串不匹配!")
```

5.3 树形关系

树形关系是一种非线性数据类型。所谓非线性数据类型，是指集合中至少存在一个元素具有多于一个的前驱或后继。表示树形关系的数据类型称为树，其定义为：树(Tree)是n(n≥0)个结点的有限集。n=0 称为空树。在任意一棵非空树中：①有且仅有一个特定称为根(Root)的结点；②当 n>1 时，其余结点可分为 m(m>0)个互不相交的有限集 T_1，T_2，…，T_n，其中每一个集合本身又是一棵树，并且称为根的子树(Sub-Tree)。从概念上理解，树其实是由有限个元素组成的具有层次关系的集合，并且除根结点外，每个子结点又可以分为多个不相交的子树。

树用来表示事物之间一对多的层次关系，即每一层上的元素可以和下一层中的多个元素相关，但只能和上一层中的一个元素相关，它像一棵倒挂的"树"。在树形关系中，各元素按其前驱和后继个数，可分为根结点、叶子结点和分支结点三类。其中，根结点没有前驱结点；叶子结点没有后继结点；除根结点以外的有后继结点的结点称为分支结点，如图 5-5 所示。



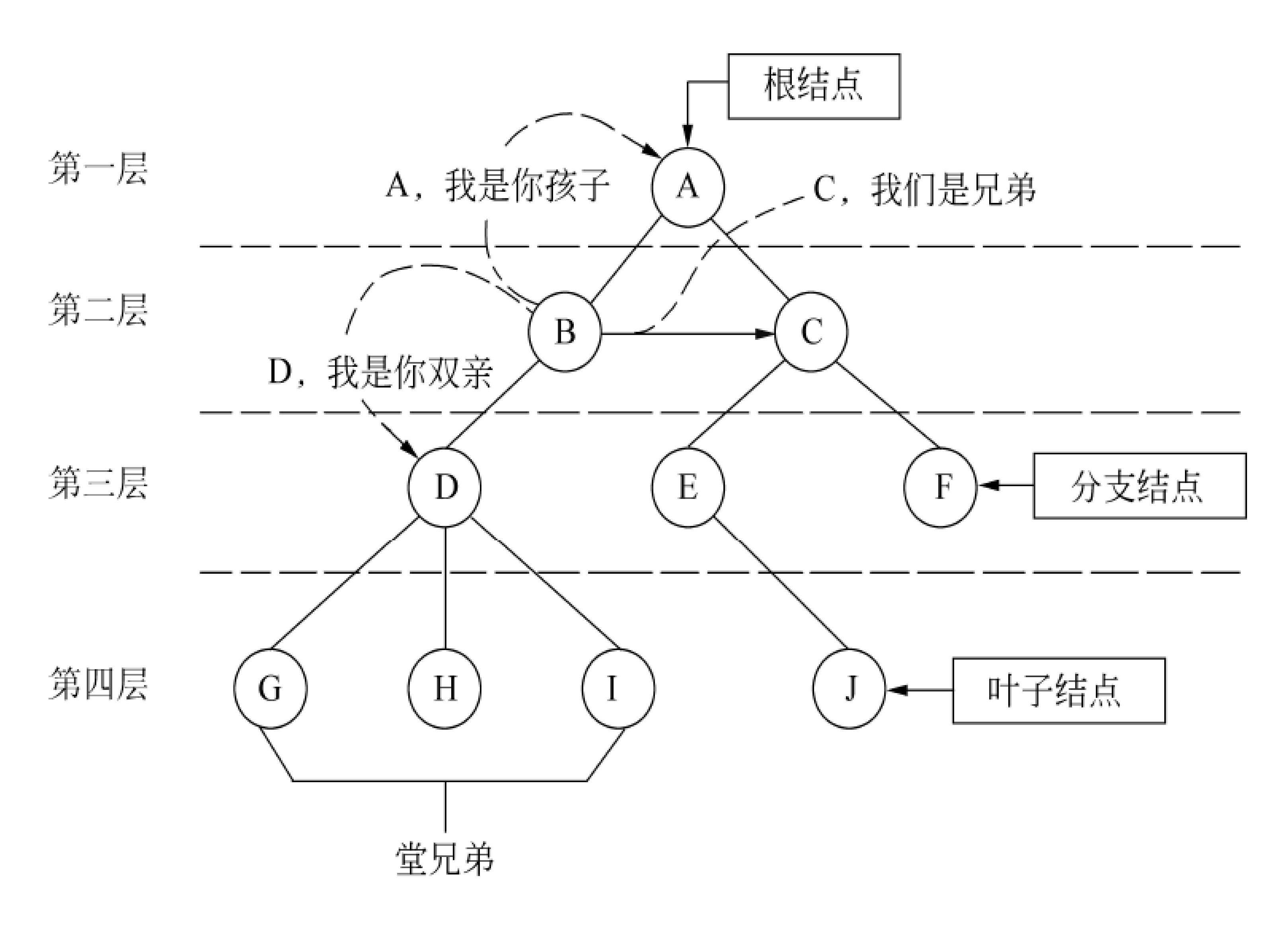


图 5-5　树中各结点之间的关系

和线性关系中各元素的地位平等不同，树形关系中的各元素因处于不同的层次而具有等级之分。直观上讲，树形关系是用分支结构来定义层次关系的。树形关系和人类社会中的家族关系十分相似，也有双亲、孩子、侄子、亲兄弟、堂兄弟、祖先、子孙等概念，如图 5-5 所示。其中，位于树顶端(第一层)的唯一一个元素称为根结点；在相邻的两层中，上一层元素(称为双亲结点)直接或间接地管理着下一层元素。在下一层中，与双亲有血缘关系的元素称为其孩子结点；否则，称为其侄子结点。一个双亲可以有 0 个、1 个或多个孩子；具有同一双亲结点的各元素间互为亲兄弟结点；双亲在同一层的各元素间互为堂兄弟；结点的祖先是指从根结点到当前结点所经过各分支上的所有结点；与之相反，结点的子孙则是该结点所在层以下的所有结点。

除上述基本概念外，树还包含很多衍生概念，例如结点的度(结点的子树个数)、树的度(树中任意结点的度的最大值)、层(根在第一层，其他结点所在层数以此类推)、结点的高度(结点所在层数)、树的高度(树中所有结点的层数的最大值)、有序树(树中各个结点是有次序的)、二叉树(每个结点最多有两个子树的树结构)、森林(多个树组成)等。了解和辨析这些概念能够帮助大家较好地理解树的特征。

树形关系在计算机领域和现实世界都有比较广泛的应用。例如，编译器中的语法树、文件中的目录结构、数据库系统中的信息组织形式、单位的组织架构、家族族谱等。

但是，计算机如何定义和存储树形关系呢？众所周知，计算机的内存是一个一维存储空间，即各元素在物理位置上仅具有线性关系，因此，我们无法直接利用这种元素间在物理上的线性存储关系来直接映射它们在逻辑上的树形关系，即无法直接使用线性表来存储树结构。我们只能通过额外存储结点间的特定关系（如双亲、孩子、兄弟等）的方式间接地表示这种具有非线性特征的数据集合。例如，在存储结点的内容同时，额外开辟一个空间来存储其双亲结点的位置，如表 5-2 所示，即人为指定元素间的相互关系。这里需要注意，计算机并不像人脑一样，能够直观地、整体地识别各元素间的树形关系，它只能通过不断判断两元素间的局部关系的方式来间接地表示所有元素间的逻辑关系。由此可见，能否正确地、简单地定义两元素间的相互关系是计算机表示非线性关系的关键。目前，表示两元素间关系的方法主要有双亲表示法（仅额外存储结点的双亲位置）、孩子表示法（仅额外存储结点的孩子位置）和孩子双亲表示法（额外存储结点的双亲位置和孩子位置）。不同的表示方法有各自的优缺点。衡量一种表示方法（即一种非线性关系）的存结构是否合理，主要取决于它对需求中各种操作的实现是否方便、高效，也就是说是否节省存储空间或计算时间。

表 5-2　树的双亲表示法

下标	Data	Parent	下标	Data	Parent
0	A	−1	5	F	2
1	B	0	6	G	3
2	C	0	7	H	3
3	D	1	8	I	3
4	E	2	9	J	4

【例 5-10】 一个单位有 10 个部门，每个部门都有一部电话，但是整个单位只有一根外线，当有电话打来时，由转接员转到内线电话，已知各部门使用外线电话的频率

(次/天)如表 5-3 所示。应该如何设计内线电话号码,使得接线员拨号次数尽可能少?

表 5-3　各部门电话使用频率

部门	部门 1	部门 2	部门 3	部门 4	部门 5
频率/(次/天)	5	20	10	12	8
部门	部门 6	部门 7	部门 8	部门 9	部门 10
频率/(次/天)	4	3	5	6	9

解答: 根据题目要求,为了使“接线员拨号次数尽可能少”,直接的想法就是让各部门所使用的内线号码的长度,尤其是使用频率高的内线号码的长度尽可能短。另外,题目还隐含了一个约束条件,即各部门间的号码必须是唯一的,不能相同。

本题属于编码问题,我们需要根据各部门电话的使用频率确定对应的号码值。通过分析问题特点,我们将编码长度和树中各结点的层数(树的结点层数是指从根结点到该结点的唯一一条路径)相关联,在保证树中各结点的层数尽量少的同时,还让频率高的编码尽可能处于树的顶部,从而获得较短的路径。这其实是哈夫曼编码问题。

提示: 哈夫曼编码(Huffman Coding)是一种经典的数据压缩算法,JPEG 就是采用哈夫曼编码进行图像压缩的。在计算机数据处理中,哈夫曼编码使用变长编码表对源符号(如文件中的一个字母)进行编码,其中变长编码表是通过一种评估来源符号出现概率的方法得到的,出现概率高的字母使用较短的编码,反之出现概率低的则使用较长的编码,这便使编码之后的字符串的平均长度、期望值降低,从而达到无损压缩数据的目的。因此,理论上,哈夫曼编码可以将数据编成平均长度最小的无前缀码(Prefix-Free Code)。

为了得到上述结果,需要完成以下几个步骤:

(1) 对各部门的编码按照从小到大的顺序排序,构成顺序集合 nodeQueue。

(2) 从顺序集合中取出当前值最小的两个元素 min_1 和 min_2,并生成新的元素 newNode,newNode 的值是 min_1 和 min_2 两元素值的和。

（3）从 nodeQueue 中删除 min_1 和 min_2，并将 newNode 按序插入到 nodeQueue 中。

（4）重复步骤（2）和步骤（3），直到 newNode 仅剩下一个元素。

上述步骤中有以下需要注意的规则：

（1）nodeQueue 必须时刻保持从小到大的顺序。

（2）总是从 nodeQueue 下标为 0 和下标为 1 的位置取出当前值最小的两个元素，并将其删除。

（3）必须将新生成的元素 newNode 插入到 nodeQueue 的合适位置，从而保证 nodeQueue 的有序特征。

本题解决问题的关键就是通过不断读取 nodeQueue 中当前值最小的两个元素来动态地构造一棵二叉树，使得该二叉树中任意结点的左孩子的值均小于或等于其右孩子的值。

具体代码如下：

```
iniQueue=[[1,5,-1],[2,20,-1],[3,10,-1],[4,12,-1],[5,8,-1],[6,4,-1],[7,3,
-1],[8,5,-1],[9,6,-1],[10,9,-1]] #构造一个二维列表，其中[1,5,-1]的第一个
元素表示部门编号，第二个元素表示电话使用频率，第三个元素表示其父结点的下
标位置，初始化为-1

nodeQueue=sorted(iniQueue, key=lambda x: x[1]) #对 iniQueue 按其每个列表元
素中的第二个值进行排序

while(len(nodeQueue) >1):
#新生成一个父结点，其值为 nodeQueue 中当前值最小的两个元素和
    node=[0,-1,-1]#新生成一个结点，其部门编号为 0，即该结点不代表实际的
部门
    node[1]=nodeQueue[0][1]+nodeQueue[1][1]#更新新生成的 node 结点，令其
值等于当前 nodeQueue 序列中值最小的两个元素和
    iniQueue.append(node) #将新生成的 node 结点插入到 iniQueue 中
    nodeIndex=iniQueue.index(node) #获取 node 结点在 iniQueue 中的下标
位置
```

```
#更新 nodeQueue 中当前值最小的两个元素的父结点位置,其父结点位置即为 node
在 iniQueue 中的下标位置
    nodeQueue[0][2]=nodeIndex
    nodeQueue[1][2]=nodeIndex
#从 nodeQueue 中将当前值最小的两个元素删除
    nodeQueue.remove(nodeQueue[0])
    nodeQueue.remove(nodeQueue[0])#注意是删除第 0 个位置的值,因为上一个
nodeQueue.remove 执行后,当前的下标 1 已经向前自动移动了一位
    nodeQueue.append(node)
    nodeQueue=sorted(nodeQueue, key=lambda x: x[1])

#输出双亲表示法构成的树
for i in range(len(iniQueue)):
    print(iniQueue[i])

#构造新的树 codeQueue,codeQueue 中各结点增加了一个字符元素,用来存储该结
点到其父结点的路径值,分别是[1,5,-1,'']
codeQueue=iniQueue
for i in codeQueue:
    i.append('')

#给每个结点到其父结点之间的路径赋值,左孩子路径标'0',右孩子路径标'1'
for i in codeQueue:
    if i[2] !=-1:
        if i[3]=='':
            findParent=0
            for j in codeQueue:
                if j[2]==i[2] and i !=j:
                    if j[1] >=i[1]:#孩子结点值小的放在父亲左侧,其路径值
为 0
                        i[3]='0'
                        j[3]='1'
```

```
                    findParent=1
                else:
                    print('j3',j)#孩子结点值大的放在父亲右侧,其路径值为 1
                    i[3]='1'
                    j[3]='0'
                    findParent=1
            if findParent==0:#只有一个孩子结点的放在父亲左侧,其路径值为 0
                i[3]='0'
    else:
        i[3]=''#根结点无路径值
print(codeQueue)

#输出每个部门的编码值
for i in codeQueue:
    if i[0]>0:#部门的编号都大于 0
        str=i[3]
        parIndex=i[2]#parIndex 得到该结点的父结点下标
        while(parIndex !=-1):
            str=str+codeQueue[parIndex][3]
            parIndex=codeQueue[parIndex][2]
        print('部门',i[0],'的电话编码',str)
```

注意:(1) sorted()函数:sorted()函数在对二维列表进行排序时,会在排序后的列表与原列表之间建立关联,即当修改排序后列表中的元素值时,也会同时更新原列表中对应元素的值。因此,在执行 nodeQueue[0][2]=nodeIndex 和 nodeQueue[1][2]=nodeIndex 操作时,iniQueue 会同时更新其对应元素的父结点下标。本程序借助 sorted()函数的这一特性,省去了在 iniQueue 中查找要更新父结点下标的元素操作。

(2) lambda 函数:Python 允许用户通过 lambda 快速定义单行函数。使用 lambda 时,需要注意以下:lambda 定义的是单行函数,如果需要复杂的函数,应该定义普通函



数;lambda 参数列表可以包含多个参数,如 lambda x, y: x+y;lambda 中的表达式不能含有命令,而且只限一条表达式。

截止到本章,我们还没有学习到面向对象程序设计中类和对象等相关概念,因此,本题采用了一种最基本的通过列表方式来构造树中结点以及表示各结点间关系的方法。例如,列表[1,5,−1]中的第一个元素表示部门编号,第二个元素表示该部门的电话使用频率,第三个编号表示该部门在树中所对应的父结点下标位置。通过创建一个包括有树中各结点的二维列表 iniQueue,实现树的构造,iniQueue 二维列表结构如表 5-4 所示。

表 5-4　例 5-10 代码中 iniQueue 二维列表结构

下标位置	部门编码	电话频率	父结点下标位置
1	1	5	11
2	2	20	16
3	3	10	14
4	4	12	15
5	5	8	13
6	6	4	10
7	7	3	10
8	8	5	11
9	9	6	12
10	10	9	13
11	0	7	12
12	0	10	14
13	0	13	15
14	0	17	16
15	0	20	17
16	0	25	17

续表

下标位置	部门编码	电话频率	父结点下标位置
17	0	37	18
18	0	45	18
19	0	82	−1

5.4 网状关系

网状关系，也称为图关系，也是一种非线性数据类型，它用来表示事物之间的多对多的复杂关系。表示网状关系的数据类型称为图，其定义为：图（Graph）是由顶点（Vertex）的有穷非空集合和顶点之间边（Edge）的集合组成的，通常表示为 G(V, E)，其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中边的集合。

在现实世界中，人与人之间的关系复杂得像一张大网，如何快速识别出来“你的姑妈的孩子的堂兄弟的姑父”和你究竟是何种关系？用线性关系或树形关系都是无能为力的。图是一种解决多对多关系的有效手段。在图中，任何元素之间都可能存在关系，即每个元素都可以有多个前驱，也可以有多个后继。但是，与树中各元素因处于不同层次而具有不同等级不同，图中各元素之间的关系却是平等的、无序的。因此，图没有层的概念。我们无法在图中找到谁是第一个顶点，也无法判断与顶点相连的其他各顶点之间的次序关系。

由于图比较复杂，所以它涉及的概念也很多。根据图中顶点和边的属性及关系，我们可以将图划分为不同的类型。例如，按照图中两点之间的边是否有方向，可以将图定义成有向图（Directed Graph）和无向图（Undirected Graph）；按照图中是否有重边或自回路（顶点到其自身的边），可以将图定义成简单图和复杂图；按照任意两点间是否均存在边，可以将图定义成完全图和不完全图；按照边是边是否带有权值（Weight），可以定义成图和网（Network）；按照边的数量多少，可以将图定义成稀疏图和稠密图。两个图 $G=(V, E)$ 和 $G'=(V', E')$，如果 $V'\subseteq V$ 且 $E'\subseteq E$，则称 G' 为 G 的子图（Sub-Graph）。



其实，我们定义图主要是为了研究顶点和边之间的关系。在无向图中，若两个顶点 v_1 和 v_2 之间有关系，则称它们之间存在一条边，用 (v_1, v_2) 表示，且 $(v_1, v_2)=(v_2, v_1)$。在有向图中，两个顶点之间的边称为弧(Arc)，用 $<v1,v2>$ 表示，且 $<v_1, v_2> \neq <v_2, v_1>$。在无向图中，与一个顶点相关联的边的数目，称为该顶点的度(Degree)。在有向图中，以一个顶点为终点的边的数目称为入度(In-Degree)，以一个顶点为始点的边的数目称为出度(Out-Degree)。在无向图 $G(V, E)$ 中，若存在一个顶点序列 $((v_0, v_i), (v_i, v_j), \cdots, (v_k, v_m))$，其中 $(v_i, v_j) \in E$，则称该序列为顶点 v_0 到顶点 v_m 的一条路径(Path)。在有向图 $G(V,E)$ 中，若存在一个顶点序列 $(<v_0, v_i>, <v_i, v_j>, \cdots, <v_k, v_m>)$，其中 $<v_i, v_j> \in E$，则称该序列为顶点 v_0 到顶点 v_m 的一条路径。路径上边或弧的长度称为路径长度。从图的概念，我们不难发现，图中边的方向性对了解图的性质十分重要，因此使用前，首先要判断图是有向的还是无向的。另外，树中根结点到任意结点的路径是唯一的，而图中顶点与顶点之间的路径却不是唯一的。这也是图十分复杂的主要原因之一。

图在现实中的应用比比皆是，只要事物之间存在着多对多的非线性关系，就会有图的身影。例如，由地点构成顶点，由连接地点的公路构成边的公路、铁路交通图；由元件构成顶点，由元件间的电子线路构成边的电路图；由用户终端的各台计算机构成顶点，由计算机间的网络连线构成边的网络拓扑图；由生产工序构成顶点，由各道工序之间的顺序关系构成边的生产流程图等。

和树形关系一样，图也是一种非线性关系。但是，它的逻辑结构比树更加复杂，图中的任意两个顶点之间都可能存在关系，且各顶点间的关系是无序的。因此，图的存储也无法直接用线性表实现，即我们无法根据图中各元素在内存中的物理位置(线性关系)来直接表示它们之间复杂的逻辑关系。计算机解决图的存储方法和树类似，也是通过额外开辟内存空间的方式来存储各顶点间的关系。图是由顶点和边构成的，而边实际上就是表示两顶点间的关系。由此可知，图的存储主要是存储各顶点位置和顶点之间的相互关系。目前，计算机主要有 5 种存储图的方法：邻接矩阵、邻接表、十字链表、邻接多重表和边集数组。本文主要介绍最基本也是最常用的邻接矩阵方法。

图的邻接矩阵表示法其实就是使用两个数组来分别存储图中的顶点信息(一维数组,如图 5-6(b)所示)和各顶点间边或弧的信息(二维数组,如图 5-6(c)所示)。对于具有 n 个结点的无向图(或有向图),其邻接矩阵定义为:

$$edge[i][j]=\begin{cases}1, & 若(v_i,v_j)\in E 或 <v_i,v_j>\in E\\0, & 其他情况\end{cases}$$

对于具有 n 个结点带权值的无向图(或有向图),即网图,其邻接矩阵定义为:

$$edge[i][j]=\begin{cases}W_{ij}, & 若(v_i,v_j)\in E 或 <v_i,v_j>\in E\\0, & 若 i=j\\\infty, & 其他情况\end{cases}$$

其中,W_{ij}表示边(v_i,v_j)或弧$<v_i,v_j>$上的权值。

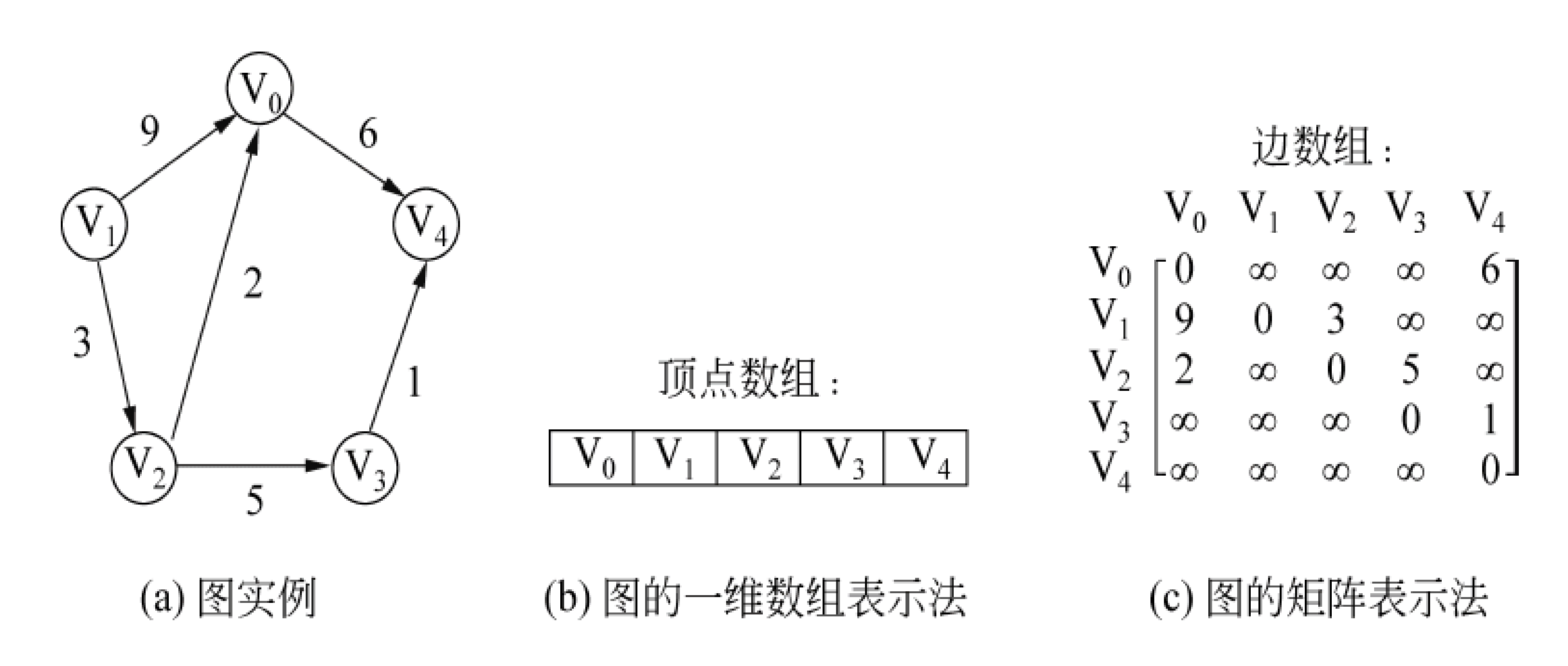


图 5-6 图的邻接矩阵表示法

【例 5-11】 假设驾车从起点 A 地出发,要到终点 E 地,沿途可以有几条路径供选择,试建立数学模型,模拟汽车导航仪选择一条行驶距离最短的行车线路(里程数据可通过百度地图获得),如图 5-7 所示。

解答:这是一道关于“最短路径”的实际问题。在分析问题时,首先判断用有向图的方法进行求解。然后,通过对问题的抽象和建模,最终实现计算机的求解。具体建模过程如下。

1) 模型抽象和假设

用计算机解决实际问题时,要对该问题进行适度、适当、适量的抽象和假设,以降低





图 5-7　百度地图示例图

求解的复杂程度。

首先，将图 5-7 的实际的行驶路线抽象为由点和线组成的简单图形，如图 5-8 所示。图中边的权值表示两点间的距离。

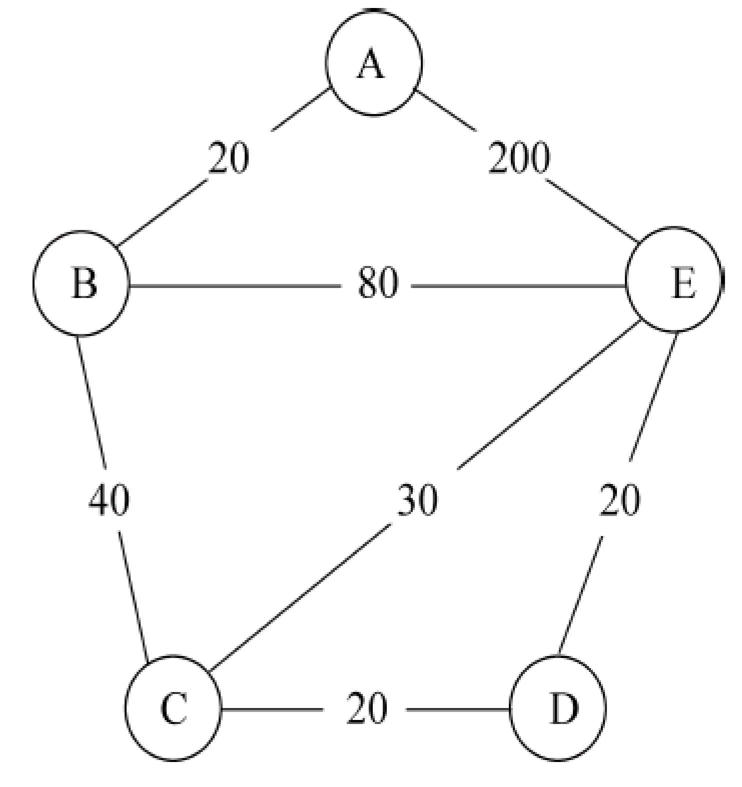


图 5-8　行驶路线简图

其次，关于“最短路径问题”，我们做以下几点假设：

(1) 假设任意两地间均有路径可达。

(2) 驾驶人完全按照导航指示的路线行驶，不走其他路线。

(3) 到访点和行驶路径不存在重复、折返等情况。

再次，我们对“最短路径问题”中出现的变量做数字化处理：

① G：表示带权有向图，$G=(V, E)$，其中 V 为顶点集合，E 为边集合。

② V_i：表示第 i 个到访地点$(i=1,2,\cdots,n)$。

③ V：表示所有到访点的集合，$V=\{V_i\}$。

④ d_{ij}：表示 V_i 和 V_j 间的距离 d_{ij}，$d_{ij}=d_{ji}(i\neq j)$。

⑤ E：表示 V_i 和 V_j 间的边，$E=\{E_i\}$，$E_i=d_{ij}$。

⑥ S：表示 V 的子集，即 $\forall S\subset V$。

⑦ $|S|$：表示集合 S 中包含的顶点个数。

⑧ $<V_i,V_j>$：表示从 V_i 到 V_j 的有向路径，$<V_i,V_j>\neq<V_j,V_i>$。

⑨ X_{ij}：$X_{ij}=\begin{cases}1, & \text{表示经过有向路径}<V_i,V_j> \\ 0, & \text{其他}\end{cases}$。

⑩ Z：表示满足题目要求的各结点间距离之和。

⑪ d(Z)：表示满足题目要求的最短距离。

2）模型建立

根据求解目的，我们建立“最短路径问题”的目标函数：$d(Z)=\min\left(\sum_{i=1}^{n}\sum_{j=1}^{n}d_{ij}X_{ij}\right)$。另外，该目标函数在求解过程中还需要满足若干约束条件：①每个顶点必须经过且仅经过一次，即该顶点的入度和出度值均等于 1，其数学表达式为：$\sum_{i=1}^{n}X_{ij}=1(j\in V)$ 且 $\sum_{j=1}^{n}X_{ij}=1(i\in V)$；② 解中不存在 子回路，即顶点的任意子集不能构成回路(当图中边的条数小于或等于顶点的个数－1时，不存在回路)，其数学表达式为：$\sum_{i\in S}\sum_{j\in S}X_{ij}\leqslant|S|-1,2\leqslant|S|\leqslant n-1$。将上述分析过程通过数学方法转换为计算机能够计算的方程组，得：

$$\text{s. t.}\begin{cases}\sum_{i=1}^{n}X_{ij}=1, & j\in V \\ \sum_{j=1}^{n}X_{ij}=1, & i\in V \\ \sum_{i\in S}\sum_{j\in S}X_{ij}\leqslant|S|-1, & \forall S\subset V,2\leqslant|S|\leqslant n-1 \\ X_{ij}\in\{0,1\}\end{cases}$$

3）模型求解

“最短路径问题”是无法利用计算机计算出精确解的。因此，当数学模型创建好后，



我们并不能按照将数学模型直接翻译成计算机程序的方式来求解问题，即该类问题的模型求解方法（算法）和模型建立方法一般是不相同的。虽然枚举方法可以按照创建的模型遍历出结果，但其算法效率很低，无法应用到实际系统中。这里介绍计算机中两种经典的求解“最短路径问题”的算法：Dijkstra 算法和 Floyd 算法。

（1）Dijkstra 算法

Dijkstra 算法是解单源最短路径问题的贪心算法，用于计算一个结点到其他所有结点的最短路径。Dijkstra 算法的主要特点是以起始点为中心向外层扩展，直到扩展到终点为止。Dijkstra 算法的核心思想是：把图 G 中顶点集合 V 分成两组，第一组为已经求出最短路径的顶点集合（用 S 表示）；第二组为其余未确定最短路径的顶点集合（用 U 表示）。初始时 S 中只有一个源点（用 v 表示），按最短路径长度的递增次序依次把第二组的顶点加入 S 中。在加入的过程中，总保持从源点 v 到 S 中各顶点的最短路径长度不大于从源点 v 到 U 中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S 中顶点的距离就是从 v 到此顶点的最短路径长度，U 中顶点的距离是从 v 到此顶点只包括 S 中的顶点为中间顶点的当前最短路径长度。

```
infi=float('inf')
def dijkstra(graph,n,toFind):
    dis=[0]*n
    founded=[False]*n
    path=[0]*n
    length=1
    founded[0]=True
    k=0
    for i in range(n):
        dis[i]=graph[k][i]
    for j in range(n-1):
        mini=infi
        for i in range(n):
            if dis[i]<mini and not founded[i]:
```

```
                mini=dis[i]
                k=i
            founded[k]=True
            for i in range(n):
                if dis[i]>dis[k]+graph[k][i]:
                    dis[i]=dis[k]+graph[k][i]
                    if i==toFind:
                        path[length]=k
                        length+=1
    path[length]=toFind
    return  path,dis[toFind]

n=5  #图中结点个数
toFind=4  #寻找到 E 的最短路径
paths=[
        [0,20,infi,infi,200],
        [20,0,40,infi,80],
        [infi,40,0,20,30],
        [infi,infi,20,0,20],
        [200,80,30,20,0]
        ]
shortestPath,shortestDistance=dijkstra(paths,n,toFind)
#shortestPath 是从 0 到 E 的最短路径经过的结点,shortestDistance 是距离
print(shortestPath)
print(shortestDistance)
```

(2) Floyd 算法

Floyd 算法是解决多源点之间(任意两点间)最短路径的动态规划算法。Floyd 算法的核心思想是：任意结点 V_i 到任意结点 V_j 的最短路径仅存在两种可能方案，要么从 V_i 直接到 V_j；要么从 V_i 间接到 V_j(经过若干个中间结点 V_k)。假设 Dis(i,j)为 V_i 到 V_j 的最短路径的距离，对于每一个中间结点 V_k，判断 $Dis(i,k)+Dis(k,j) < Dis(i,j)$ 是否成立：



如果成立，则 Dis(i,j)＝Dis(i,k)＋Dis(k,j)。当我们遍历完所有中间结点 k，Dis(i,j)中的记录是从 V_i 到 V_j 的最短路径的距离。

```
N=5 #图中结点个数
toFind=4
infi=float('inf') #无穷大
graph=[
        [0,20,infi,infi,200],
        [20,0,40,infi,80],
        [infi,40,0,20,30],
        [infi,infi,20,0,20],
        [200,80,30,20,0]
    ]
path=[[-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1],[-1,-1,-1,
-1,-1],[-1,-1,-1,-1,-1]]
def back_path(path,i,j): #递归回溯,找最短路径
    if path[i][j]!=-1:
        back_path(path,i,path[i][j]) #比如 0~4 中的一个点是 3,先递归找 0~3
的最短路径,再输出 3,最后输出 3~4 的最短路径
        print (path[i][j])
        back_path(path,path[i][j],j)
    return;
for l in range(0,N):
    for m in range(0,N):
        for n in range(0,N):
            if graph[m][n]>graph[m][l]+graph[l][n] and m!=n and graph[m]
[l]!=infi and graph[l][n]!=infi:
                graph[m][n]=graph[m][l]+graph[l][n]
                path[m][n]=l
print ("最短距离:")
print (graph[0][4])
print ("经过的路径:")
```

```
print (0)
back_path(path,0,toFind);
print (toFind)
```

习　　题

1. 有面值为1元、2元和5元的纸币,如何得到最少的找零数目?

2. 模拟理发店。假设:

(1) 理发店共有3名理发师,他们的编号分别为0号、1号和2号。

(2) 每位理发师剪一个头发的时间都是1小时。

(3) 理发师按编号顺序依次给顾客剪头发。

(4) 顾客们都是很有时间观念的人而且非常挑剔,他们对于每次光顾理发店时所能容忍的最长等待时间是3小时,而且等待时间越长,顾客的满意度越低。如果3小时还不能轮到自己剪头发,顾客会立刻生气离开。

(5) 一天中各时间段顾客人数如表5-5所示,每个顾客的编号从0开始依次递增1,如9点为0号,10点为1号和2号等。

表5-5　各时段顾客人数(人)

时间	9:00	10:00	11:00	12:00	13:00	14:00	15:00
人数	1	2	4	3	6	2	3
时间	16:00	17:00	18:00	19:00	20:00	21:00	22:00
人数	4	5	7	10	8	3	1

请输出:

(1) 每位顾客所对应的理发师编号。

(2) 每位顾客开始剪发时间、等待时间、离开时间。

(3) 若顾客没有被服务,则输出该顾客的进店时间、等待时间和离开时间。



3．用栈实现四则运算，计算如 9＋(3－1)×3＋10/2 的值。

提示：本题分两步实现：①将数学表达式表示为后缀表达式；②用栈实现后缀表达式的计算。

4．用栈实现十进制正整数到二进制的转换。

5．设有 100 个学生的“Python 程序设计”课程的考试成绩公布如表 5-6 所示。

表 5-6　学生成绩数据分布情况表

分数	0～59	60～69	70～79	80～89	90～100
人数比例	5%	15%	40%	30%	10%

请编写程序，要求根据用户输入的每个学生的考试成绩，快速地找到其对应的成绩等级，并将其打印出来。

6．请编写程序实现约瑟夫斯问题。所谓约瑟夫斯问题(Josephus Problem)，是指参与者围成一个圆圈，从某个人(队首)开始报数，报数到 n＋1 的人退出圆圈，然后从退出人的下一位重新开始报数；重复以上动作，直到只剩下一个人为止。

7．社团管理，要求编写程序实现以下功能：①社团招收新成员；②修改社团相应信息；③老成员离开社团；④查询社团情况；⑤统计社团成员数。

8．地铁最少换乘问题。已知有两条地铁线路，其中 A 为环线，B 为东西向线路，线路都是双向的。经过的站点名分别如下表 5-7，两条线交叉的换乘点用 T1、T2 表示。请编写程序，要求任意输入两个不同的站点名称，输出乘坐地铁最少需要经过的车站数量(含输入的起点和终点，换乘站点只计算一次)。

表 5-7　地铁线路表

线路名称	经过的站名									
地铁线 A(环线)	A1	A2	A3	A4	A5	A6	A7	A8	A9	T1
	A10	A11	A12	A13	T2	A14	A15	A16	A17	A18
地铁线 B(直线)	B1	B2	B3	B4	B5	T1	B6	B7	B8	B9
	B10	T2	B11	B12	B13	B14	B15	—	—	—



第6章　程序编写方法

程序是计算机解题的手段。对于一个简单的问题，通过直接逐条编写程序能够获得问题的求解。但是，随着问题的复杂性增加，所编写的程序的规模和复杂度也相应地不断增加，逐条编写程序的方法显然不能满足人们对程序的设计要求。

20世纪60年代末，面向过程的程序设计方法（又称结构化方法）出现了，它提高了语言的抽象层次，程序中采用具有一定含义的数据名称和简单的语句，使程序可以更好地与所描述的事物联系起来。面向过程的编程方法采用自顶向下、逐步求精、模块化等设计思想，将问题分解为一个个子问题，这些问题可以由一个人或者多个人解决，从而提高了速度，并且便于对程序进行调试，有利于软件的开发维护。由于这些优点，面向过程的程序设计方法迅速得到了人们的认可，然而在面向过程的程序设计中，数据结构和操作过程是两个相互独立的实体，数据结构的变化会引起数据操作的变化，给程序的维护造成了很大的困难。面向对象的设计方法应运而生，它借鉴了面向过程的一些设计方法，将软件系统分解为一个个对象，对象由编程者从现实世界物体中抽象出来，由数据及对数据操作过程构成。面向对象的编程方法继承了面向过程程序设计的一些优点，同时将数据和操作封装在对象里，使编程人员可以将数据和操作联系在一起，有效解决了面向过程编程中程序维护和调试的困难。

在本章的最后还介绍了模块、文件等内容，编程时合理地利用这些工具会使编程达到事半功倍的效果。本章在介绍程序设计方法的同时，还穿插一些具体的小例子，以帮助理解。

6.1　逐条编程

【例 6-1】　编写程序绘制一个如图 6.1 所示的三角形。

逐条程序编写打印：

```
print("*")
print("* * *")
print("* * * * *")
print("* * * * * * *")
print("* * * * *")
print("* * *")
print("*")
```

```
*
***
*****
*******
*****
***
*
```

图 6-1　三角形

可以看到，虽然将这个三角形打印出来了，在编写的程序中却使用了 7 个 print 语句，假如现在需要打印一个更大或者更小的三角形，则需要重新输入一遍多条 print 语句，造成大量的重复输入。假如我们不想使用 * 而使用其他符号，如何输出该三角形呢？可以发现，编程的过程中存在着语句的重复输入，代码的冗余也给程序的扩展和维护造成了极大的困难。

像例 6-1 这样一条一条地编写程序并逐条输出结果就是逐条编程。在默认状态下，程序由一条条指令组成，按照从上到下顺序执行。这种方法思路简单，但并不实用。

换一种思路，可以如下定义一个输出过程(函数)用来打印上述图形：

```
def draw(n,c):
    for x in range(1, 2*n, 2):
        print c * x
        if x==2*n-1:
            for x in range(2*n-3, 0, -2):
                print c * x
>>>draw(4,'*')#调用这个打印图形的过程
```

使用函数 draw( )可以指定打印所需的字符和打印图形的行数,这样每次打印不需要重新编写 print 函数,只需通过函数调用来实现,并且需求的变更容易实现。例如,打印一个更大的三角形或打印一个用“#”构成的三角形等。使用函数能够有效地避免大量重复的代码,降低代码的冗余度。

6.2 面向过程编程

面向过程编程是一种以过程为中心的编程方法。当遇到复杂问题一时无从下手时,可以采取自顶向下、逐步求精的方法,即将复杂问题逐层拆分为若干个小问题,直至每个小问题都可以用较为简单的算法实现。在划分过程中,首先要注意各个问题尽量保持独立,避免程序冗余。

面向过程的编程方法,首先分析出解决问题所需要的步骤,然后把这些步骤用函数一一实现,使用的时候一个一个依次调用,调用函数时需要给出必要的参数。本节除了介绍函数,还会详细介绍参数和作用域的概念。

6.2.1 函数

在编写较大型的应用程序时,若在一个地方用到一种功能(如对数据的输出),在另一个地方也需要用到这样一种功能,则可以在所有需要的地方都编写同一段代码,为了避免代码的冗杂,可以将该段代码组织为一个特定的语句组来完成相应的功能,这组语句可以作为一个单位使用,我们称这个语句组为函数,并指定函数名。通过使用函数名可以在程序的不同地方多次执行这组语句,却不需要在所有地方都重复编写该语句组,这一过程称为函数调用。

在 Python 中,有些函数是系统自带的,即 Python 中的标准函数,也称为内置函数。还有些函数是第三方编写的,即由其他程序员编写的一些函数,我们称这些函数为第三方函数。对于这些现成的函数用户可以直接拿来使用。另外,有一类函数是用户自己编写的,通常称为自定义函数。

Python 内置了一系列的常用函数供编程者使用。在编写程序时直接使用函数名(参数)的方式调用 Python 内置函数。表 6-1 列举了一些常用的内置函数及其含义。

表 6-1　Python 内置函数及其含义

函　　数	描　　述
input()	input 函数从 sys. stdin 接受原始输入并返回字符串
len(s)	len()函数返回序列(字符串、元组或列表)或字典对象的长度
list(seq)	list()函数返回列表
max(s,[,args…])	max()函数返回最大值
min(s[,args…])	当仅给定一个参数时,min()函数返回序列 s 的最小值
help(function_name)	在交互式解释器中使用就可以得到帮助函数,包括它的文档字符串信息

同时,Python 还提供了很多标准函数库用于完成很多通用的任务,除了刚刚介绍的内置函数外,还有很多库函数则放在模块下的文件中,这些模块在安装 Python 时已经一并复制,需要调用这些模块内的库函数,需要使用 import 语句导入模块或者模块所在包。

【例 6-2】　库函数调用示例。

pow(x,y[,z])是 Python 中常用的一个函数,它返回以 x 为底,y 为指数的幂。如果给出 z 值,该函数就计算 x 的 y 次幂值被 z 取模的值。试着输入下面语句:

```
>>>import math
>>>print (pow(4,2))
16
>>>print pow(4.2,3)
1
```

可以看到 4 的 2 次幂为 16,而当给出 z 值为 3 后输出取模后的值为 1(16 mod 3)。

有时候 Python 自带的函数不能完全满足人们的需求,这时候就需要定义自己的函数。Python 规定自定义函数代码块以 def 关键词开头,后接函数标识符名称和圆括号“( )”,任何传入参数和自变量必须放在圆括号中间,函数的第一行语句可以选择性地使用文档字符串对函数作说明。函数内容以冒号开始,函数体缩进,形式如下:

```
def function_name(arg1,arg2[,…]):
    statement
[return value]
```

其中,function_name 为所要定义的函数名,Python 规定函数名必须以下画线或字母开头,可以包含任意字母或下画线,函数名也不能是保留字,并且区分大小写;arg1, arg2[,…]为形式参数,函数可以有多个形式参数,形式参数之间用逗号分开。

定义函数时需要注意以下两点:

(1) 函数定义必须放在函数调用前,否则编译器会由于找不到该函数而报错。

(2) 返回值不是必需的,如果没有 return 语句,则 Python 默认返回值 None。

【例 6-3】 求 1×2×3…×n 的结果。

```
def factorial(n):                #定义阶乘函数
        result=n
        for i in range(1,n):
          result*=i
          return result          #返回 result
>>>n=3
>>>result1=factorial(n)          #函数调用
>>>print(result1)                #输出结果
```

阶乘在数学里经常用到,从例 6-3 可以看出,我们使用函数名(参数)的形式调用函数。函数的入口称为参数,Python 中的参数可以是变量,也可以是表达式;参数可以没有,也可以有一个或者多个。函数的出口则称为函数的返回值,与函数的参数一样,函数的返回值可以没有,也可以有一个或者多个。若有返回值,则在函数结束时有 return 语句。

函数体通过程序的函数调用指令来执行,程序运行到函数调用处暂停执行,把调用函数时的实参赋值给形参,执行所调用函数,在函数执行结束后,返回暂停处继续往下执行,以例 6-3 为例,其详细过程如图 6-2 所示。

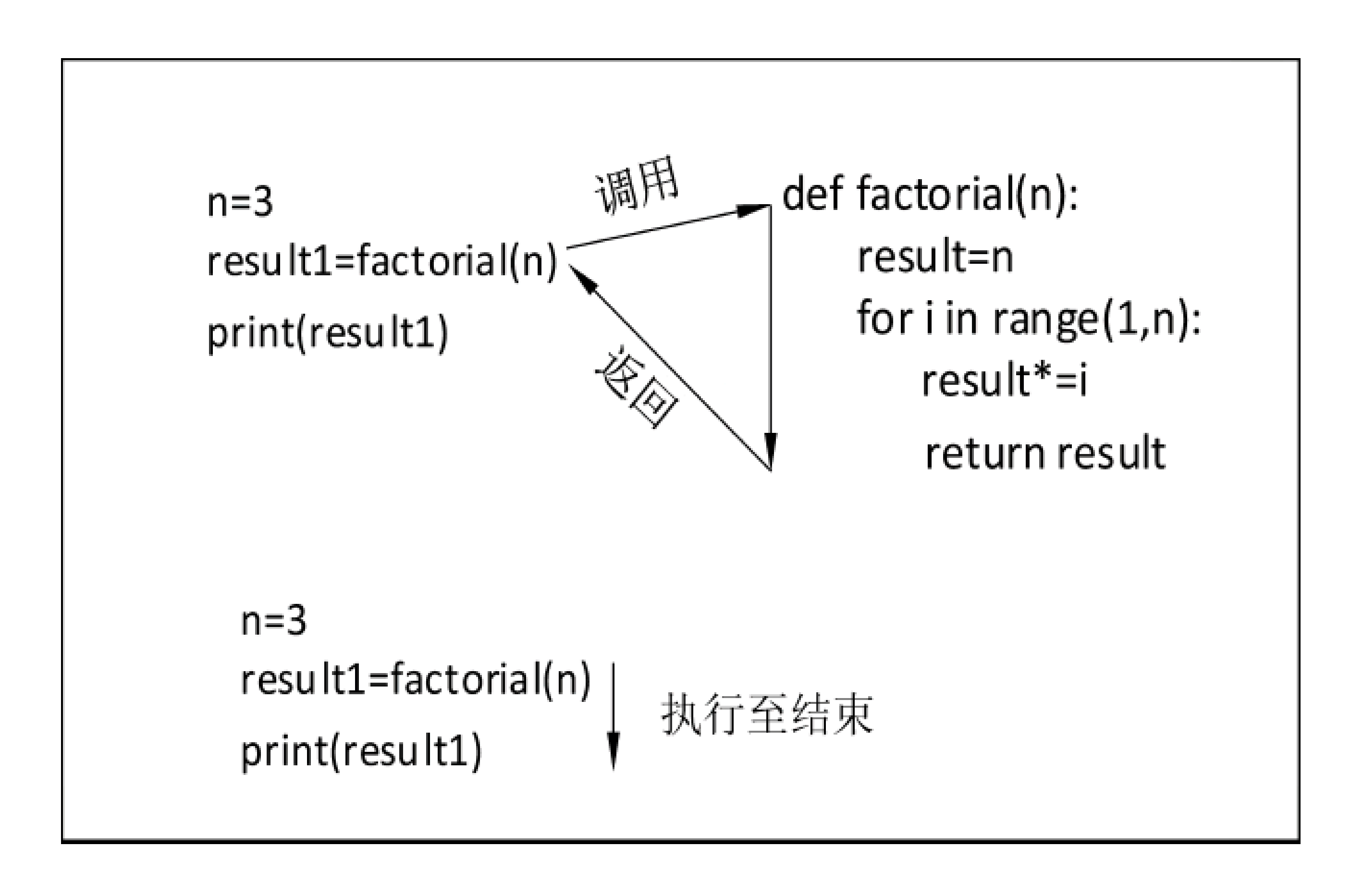


图 6-2　函数调用过程示意

首先,程序从 n=3 开始执行,当程序执行到 factorial(n)时,程序跳转的 factorial(n)函数处执行,当执行结束后返回,执行 print 函数。

当函数内部需要继续调用其他函数时称为多重调用,此时程序的执行过程为:当带有函数的程序在执行时要从函数外的第一条指令处顺序执行,遇到函数调用时,转向被调用函数名处执行,当所调用函数内部有其他函数时,同样执行以上规则,执行结束后返回被调用处继续顺序执行。例如,求函数 x^2+5 的值,首先可以定义一个求平方的函数 square(),然后求出函数 f(x)的值,通过 main()函数调用,这就是一个很典型的多层调用过程。

【例 6-4】　编写代码求函数 x^2+5 的值。

```
def square(n):
    t=n*n
    return t
def f(n):
    c =5
    res =c+square(n)
    return res
def main():
```

```
n = int(input('输入 x 值:'))
r = f(n)
print( 'The result is:',r)
```

其执行顺序如图 6-3 所示。

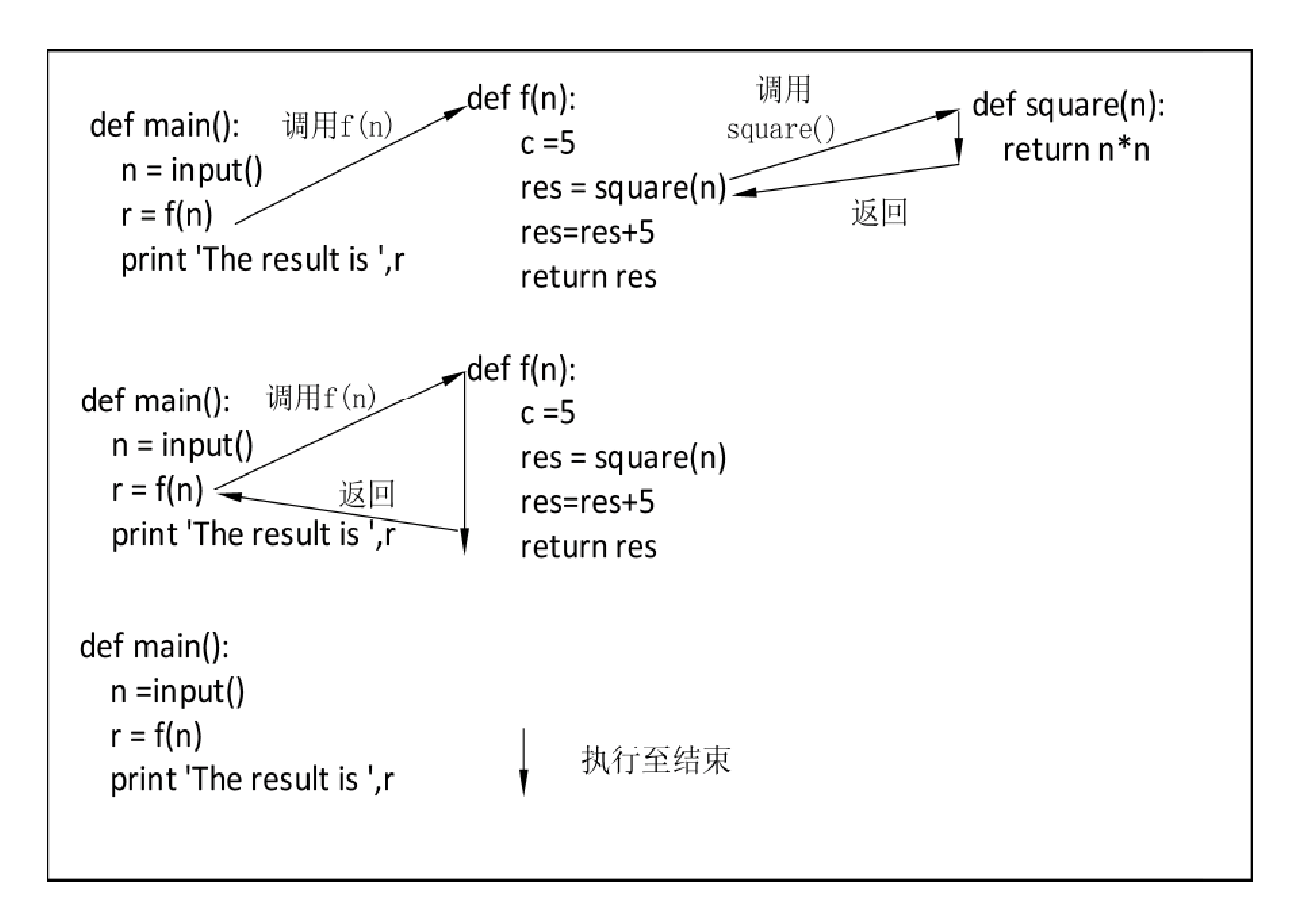


图 6-3　函数多重调用过程示意

从上面例子可以看出，当函数多层调用时，函数的调用过程仍和一层调用时的过程相同。函数调用自己是合法的，我们称函数调用自身的行为为递归。

先来看如下假设的一个函数定义：

```
def recursion():
    return recursion()
```

在个人计算机上执行上述程序可能会发生异常，从理论上讲它会永远执行下去，每次执行程序耗费一点内存，最终程序以“超过程序最大递归深度”的错误信息结束。我们把这类递归称为无穷递归，类似于 while true 开始的无穷循环。这种递归函数是没有意

义的,可以使用的递归函数应该在函数有返回值时有基本实例(即函数可以跳出),并且包含一个或多个最小问题的最小部分递归调用。递归函数的最关键的是要将问题分解为最小部分。

假如需要计算整数 x 的 n 次幂,可以用 Python 内置的 pow 函数,它需要乘以自身 n－1 次。如果自己写一个函数来计算 x 的 n 次幂要怎么写呢?先来看如下非递归实现过程:

```
def myPow(x,n):
    res =1
  for i in range(n):
    res * =x
    return res
```

程序很简单,若用递归的方法来实现呢?首先,注意到一个运算过程:对于任意的 x 来说,power(x,0)＝1;对于任意 n＞0,myPow(x,n)＝myPow(x,n－1)。理解该过程之后,程序很容易写出:

```
def myPower():
    if n==0:
        return 1
    else:
        return myPow(x,n) * =myPow(x,n-1)
```

假设现在要求计算 myPow(2,3),其执行过程如图 6-4 所示。

可以看到程序先一步一步地调用,当 n＝0 时,返回值为 1。然后,一步步地返回,直到最后函数结束,即可求得结果为 8,这样就完成了计算幂函数的递归写法。递归的理解比较困难,实现起来却没有那么复杂,要多练习有关递归的题目才能掌握它。

6.2.2　参数

函数被定义后,我们在使用它时还需要传递其所需操作的值,即参数。在 6.2.1 节




给出了函数的定义方法：

```
def function_name(arg1,arg2[,…]):
    statement
    [return value]
```

在上面定义中，arg1，arg2[，…]即为函数所需的参数，写在 def 语句函数后面的变量通常称为函数的形式参数，而调用函数时所提供的值是实际参数或者简称为参数。

这里需要注意的是，参数只是变量而已，在函数内为参数赋值不会改变任何外部变量的值，但是在一个函数中改变一个可变的对象参数会影响调用者，例如列表、字典、数组等。参数是对象指针，无须定义传递的对象类型。

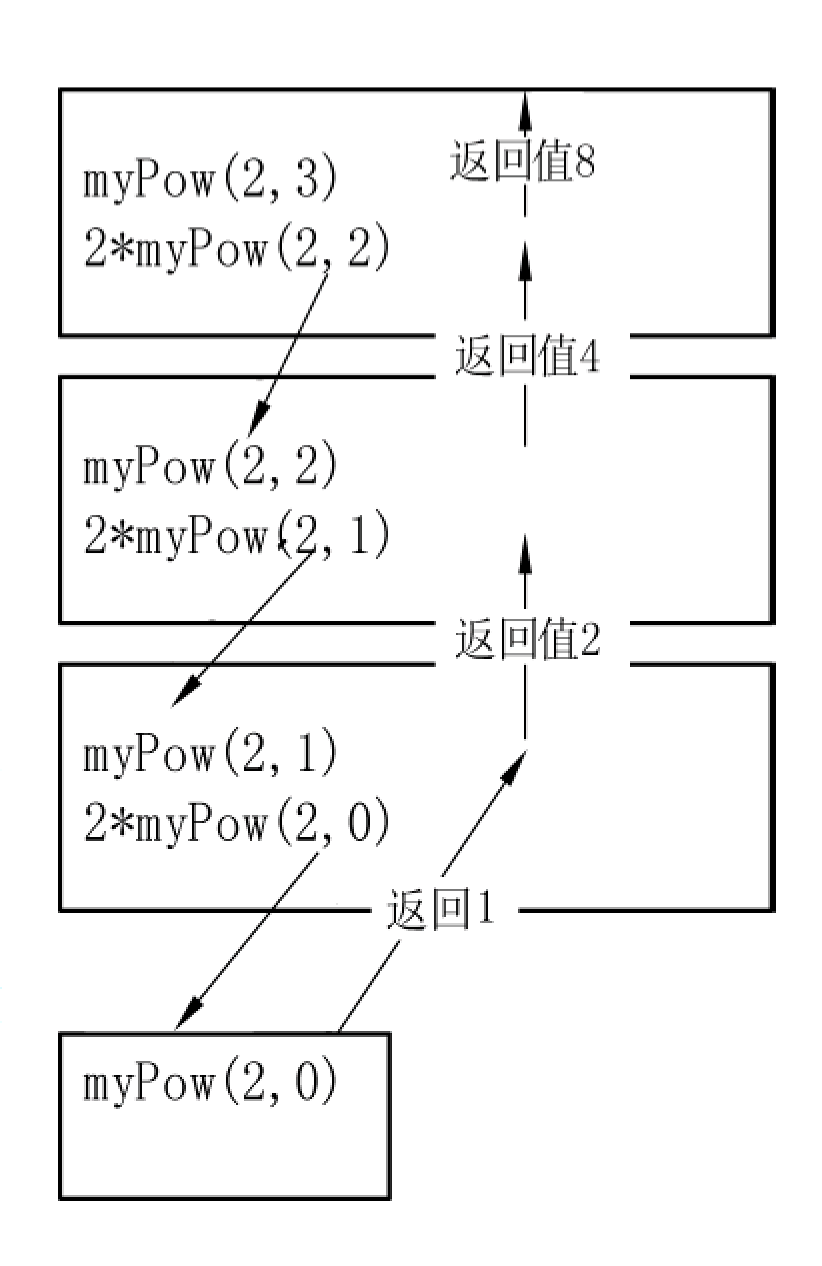


图 6-4　递归函数的执行过程

【例 6-5】 定义函数求不同类型 a+b 的值。

```
def test(a,b):
    return(a+b)

>>>r1=test("1","2")
>>>r2=test(1,2)
>>>r3=test([1],[2])
>>>print(r1)
>>>print(r2)
>>>print(r3)
```

结果分别为：

```
12
3
[1, 2]
```

可以看出，对于函数 test(a,b)来说，无论输入的参数是什么类型，该函数均能正确运行，变量 a 和 b 称为形式参数，而实际参数分别是字符串、整数和列表类型。相应地，加号"+"在此处也被赋予了不同的含义，分别表示字符串的连接、整数的加法和列表的合并。

6.2.3　作用域

Python 使用名称空间的概念存储对象，这个名称空间就是对象作用的区域，作用域就是起作用的范围，对变量的访问权限决定于这个变量是在哪里被赋值的。不同对象存在于不同的作用域。

在 Python 中，变量名引用分为以下 4 个作用域进行查找。

(1) L：local，局部作用域，即在函数内定义的变量。

(2) E：enclosing，嵌套作用域，即包含此函数的上级函数的局部作用域。

(3) G：global，全局作用域，即模块级别定义的变量。

(4) B：built-in，内嵌作用域，即系统固定模块里面的变量，如 int 等。

搜索变量的优先级顺序是：局部作用域>外层嵌套作用域>当前模块中的全局作用域>Python 内嵌作用域(即 LEGB)。

【例 6-6】　函数的作用域示例。

分析下面的例子，仔细体会变量 x 值的变换过程。

```
def f(x):                    #定义函数 f(x)
        y=1
        x=x+y
        print('x=',x)        #显示 x 的值
        return x             #返回 x 的值
>>>x=4
>>>y=2
>>>z=f(x)
>>>print('z=',z)             #显示 z 的值
```





```
>>>print('x=',x)                #显示 x 的值
>>>print('y=',y)                #显示 y 的值
```

可以看到,结果分别输出了 5、4、2 的值。也就是说,在计算 z 时是在 f(x)内部计算,此时 y=1;而当输出 y 值时是在函数 f(x)外部,此时 y=2。

有时候希望在局部作用域中改变全局作用域的对象,这时候就必须使用 Python 提供的 global 关键字去改变变量的作用域。

【例 6-7】 仔细分析下例子 x 的值变化过程。

```
def func():
        global x
        print 'x is', x
        x =2
        print 'Changed local x to', x

>>>x =5
>>>func()
>>>print 'Now the value of x is', x
```

这里将变量 x 在函数内部定义成 global 类型,即声明 a 是全局的,当我们在函数内对 x 进行操作也会修改函数外面的 a,执行上面函数可以看到,第一条 print 语句输出的 x 的值为 5,第二条 print 语句输出的 x 的值为 2。

6.3 面向对象编程

6.2 节介绍了面向过程编程,在面向过程的程序设计中,数据和对数据的操作是分离的,面向过程中的过程其实就是数据被处理和被操作的过程。对于功能相对简单或者说整个执行过程清晰明确的需求,采用面向过程的程序设计是很好的选择。但是,面向过程的程序设计也存在一些问题,例如不易修改、不容易扩展、程序的可重用性差等。单纯的面向过程的程序设计方法已经不足以应对越来越复杂的程序。因此,人们提出了一种

新的程序设计方法——面向对象编程，把现实世界中的实体抽象成对象，例如一个点、一个人、一辆车等，每个对象都有属于自己的标识、状态和行为等。在面向对象程序设计中，数据和对数据的操作是结合的，因此，对象是一个既具有数据（事物的静态属性）也具有操作（事物的动态行为）的实体。对象是一类特殊的数据，它自己掌控对自己所存储数据的处理方法。外部只能使用对象预先设计好的各种处理方法，但不能随心所欲地定义新的处理方法。外部通过向对象发送一个操作请求（即“消息”）来申请对对象的操作，当对象审核并确定该操作能够响应时，就执行并以消息形式向外部返回结果。消息构成了对象与外部进行交互的界面，也称之为“接口”。外部只能通过这个接口和对象打交道。面向对象程序设计在编写程序时使用对象进行编程，实体之间通过对象之间相互作用。从被提出到现在，面向对象编程方法已经成为主流的编程方法，极大地提高了开发效率。封装、继承和多态是面向对象编程方法的三大特性。

6.3.1　类

在面向对象编程的方法中，类和我们所知道的“种类”和“类型”含义相近，它是具有相同属性和行为的一类对象的集合，为属于该类的全部对象提供抽象的描述。如果把一切都视为对象，类就是某一类对象的属性和行为的抽象。我们见过各种各样的自行车，它们都有两个车轮、脚蹬及刹车等部件，虽然它们是不同的对象，但是它们却可以用一个抽象的自行车类 bike 来表示，任何对象都是其所属类的实例。

Python 中创建一个类的方法如下：

```
class 类名:
    def __init__(self [,arg2,…]):
    def 方法名(self [,arg2,…]):
    def __del__(self):
```

self 是类中的特殊参数，它必须是任何方法的第一形式参数，不能省略，__init__和__del__方法是类中的内置方法。__init__方法在初始化数据隐式调用时执行，实例化对象时被自动调用，用来初始化对象的静态属性。__del__方法在对象被销毁时自动调用，

新的程序设计方法——面向对象编程，把现实世界中的实体抽象成对象，例如一个点、一个人、一辆车等，每个对象都有属于自己的标识、状态和行为等。在面向对象程序设计中，数据和对数据的操作是结合的，因此，对象是一个既具有数据（事物的静态属性）也具有操作（事物的动态行为）的实体。对象是一类特殊的数据，它自己掌控对自己所存储数据的处理方法。外部只能使用对象预先设计好的各种处理方法，但不能随心所欲地定义新的处理方法。外部通过向对象发送一个操作请求（即“消息”）来申请对对象的操作，当对象审核并确定该操作能够响应时，就执行并以消息形式向外部返回结果。消息构成了对象与外部进行交互的界面，也称之为“接口”。外部只能通过这个接口和对象打交道。面向对象程序设计在编写程序时使用对象进行编程，实体之间通过对象之间相互作用。从被提出到现在，面向对象编程方法已经成为主流的编程方法，极大地提高了开发效率。封装、继承和多态是面向对象编程方法的三大特性。

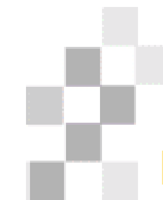
6.3.1　类

在面向对象编程的方法中，类和我们所知道的“种类”和“类型”含义相近，它是具有相同属性和行为的一类对象的集合，为属于该类的全部对象提供抽象的描述。如果把一切都视为对象，类就是某一类对象的属性和行为的抽象。我们见过各种各样的自行车，它们都有两个车轮、脚蹬及刹车等部件，虽然它们是不同的对象，但是它们却可以用一个抽象的自行车类 bike 来表示，任何对象都是其所属类的实例。

Python 中创建一个类的方法如下：

```
class 类名:
    def __init__(self [,arg2,…]):
    def 方法名(self [,arg2,…]):
    def __del__(self):
```

self 是类中的特殊参数，它必须是任何方法的第一形式参数，不能省略，__init__和__del__方法是类中的内置方法。__init__方法在初始化数据隐式调用时执行，实例化对象时被自动调用，用来初始化对象的静态属性。__del__方法在对象被销毁时自动调用，



以释放对象所占用的资源。这两个方法在类的定义中是可选的，如果不提供，Python 会提供默认的构造函数。类中的普通方法只能显示调用的执行，即只能由对象发送消息。以下是一个自定义类的例子。

【例 6-8】 我们每个人都有名字和身份证号，那么试着构造一个 Person 类，包含姓名 name 和身份证号 pid 两个属性。

```
class Person():
    def __init__(self,name,pid):
    self.name =name
    self.pid =pid
def info(self):
    print(self.name,'id is ', self.pid)
```

在 Person 类中，__init__用来初始化对象的姓名和标识。可以使用 info 来打印该对象的信息。当需要创建一个新的对象时只需要调用相应的方法即可，并不需要我们关心方法内的参数是怎么赋值的，也不用关心方法是如何实现的，这就是封装的思想，即将实现的细节隐藏，而暴露出公有接口，增强了程序的可重用性和可维护性。

类中定义的变量构成类的属性，类中变量可以分为以下类变量和实例变量：

(1) 类变量是指在类中方法外创建的变量，它在整个类有效，创建的各实例共享，可以通过“类名. 变量名”来访问该变量。

(2) 实例变量是指在方法内创建的变量，这类变量只在方法内有效，是创建的各个实例所独有的。实例变量又可以分为实例公有变量、实例私有变量、实例局部变量三种类型。

① 实例公有变量：self. 变量名，实例直接访问。

② 实例私有变量：self. __变量名，实例间接访问，格式为“实例. _类名__变量名”(注意，私有变量名前为双下画线，而类名前为单下画线)。

③ 实例局部变量：变量名，只能在函数内部访问。

以下例子说明了类变量和实例变量的定义及使用，代码中有部分需要读者自己填

充，以便加深理解。

【例 6-9】 假设圆心在点(1,2)，给定出圆心外的任一点，求出圆心的半径和周长。

```
import math
class MyPoint():
    x0 =1                                               #类变量
    y0 =2
    def __zinit__(self,x,y):
        self.x =x                                       #公有实例变量
        self.y =y
        self.__radius =math.sqrt((self.x -MyPoint.x0) * * 2+ (self.y -
       MyPoint.y0) * * 2)
        area =math.pi * math.pow(self.__radius,2)       #局部变量
        self.cir =2 * math.pi * self.__radius
                                                #私有实例变量,注意是长下画线
        ______________________          #填写 print 函数,输出圆心、半径、面积和周长
```

空格部分应该填入如下语句：

```
>>>print(p1._MyPoint__radius)
>>>p =MyPoint (4,5)
>>>print(MyPoint.x0, MyPoint.y0)
>>>print(p1.cir)
```

读者可以先试着自己补充上面的例子，注意不同类型变量的调用方法，以此来加深对类中变量的理解。

Python 中类的方法有 4 种：实例方法、内置方法、静态方法和类方法，详细介绍如下：

(1) 实例方法的定义方法为“def 方法名(self [,arg2,…])”，定义方法时默认第一个参数为 self，调用时用“对象名. 方法名([arg,…])”。

(2) 内置方法是 Python 内自带的特殊方法，例如__init__()方法、__del__()方法等，Python 中的内置方法会在下面的表格中给出。

(3) 静态方法用@staticmethod 定义且方法中不带参数,类和实例均可使用但不常用,可以用“类名.静态方法()”或者“对象名.静态方法()”来调用,在访问本类的成员时,只允许访问静态成员(即静态成员变量和静态方法),而不允许访问实例成员变量和实例方法。实例方法则无此限制。

(4) 类方法用@classmethod 定义且最少带一个类参数 cls,类和实例均可使用,可以用“类名.类方法([arg])”或者“对象名.类方法([arg])”来调用,类方法可以被对象调用,也可以被实例调用。传入的都是类对象,主要用于工厂方法,具体的实现就交给子类处理。

【例 6-10】 仔细分析下面的程序示例,体会不同变量和方法的用法。

```
class ball(object):
    count =1                          #类变量
    color = 'red'
    def __init__(self):               #内置方法
        print (self.color,self.count)
    def add_foo(self):                #实例方法
        ball.count+=1
      print (ball.count)
    @classmethod
    def foo(cls):                     #类方法
        print cls.color
        print ('calling this method foo()')
    @staticmethod
    def pish_color():                 #静态方法没有实例参数 self
        ball.color = 'blue'
        print(ball.color)
```

对上面的各种方法进行调用,其结果如下:

```
>>>a=ball()
('red',1)
```

```
>>>a,add_foo()
2
>>>a.foo()
red
calling this method foo()
>>>ball.foo()
red
calling this method foo()
>>>a.pish_color()
blue
>>>ball.pish_color()
blue
>>>a.__init__()
('blue',2)
>>>ball.add_foo()

Traceback (most recent call last):
  File "<pyshell# 28>", line 1, in <module>
    ball.add_foo()
TypeError: unbound method add_foo() must be called with ball instance as
first a rgument (got nothing instead)
```

可以看到，执行 ball.add_foo()时会报错，因为它是实例方法只能通过对象调用，而类方法和静态方法则不必必须通过对象调用，还可以看到内置方法__init__会在对象创建时自动执行，也能手动调用。

Python 中有很多封装好的类，这些类内部都有自己的内置方法，以常见的 list 方法为例，它内部有很多内置方法。例如，__delitem__(self, y)、__eq__(self, y)、__getattribute__(self, name)、__ge__(self, y)、__gt__(self, y)、__setitem__(self, i, y)等。表 6-2 给出了类中常见的内置方法及其含义。

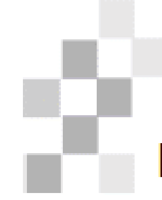


表 6-2 Python 类中常见的内置方法及其含义

内置方法	描述
__init__(self,…)	初始化对象,在创建新对象时调用
__del__(self)	释放对象,在对象被删除之前调用
__new__(cls, * args, * * kwd)	实例的生成操作
__str__(self)	在使用 print 语句时被调用
__getitem__(self,key)	获取序列的索引 key 对应的值,等价于 seq[key]
__len__(self)	在调用内联函数 len()时被调用
__cmp__(stc,dst)	比较两个对象 src 和 dst
__getattr__(s,name)	获取属性的值
__setattr__(s,name,value)	设置属性的值
__delattr__(s,name)	删除 name 属性
__getattribute__()	__getattribute__()功能与__getattr__()功能类似
__gt__(self,other)	判断 self 对象是否大于 other 对象
__lt__(slef,other)	判断 self 对象是否小于 other 对象
__ge__(slef,other)	判断 self 对象是否大于或者等于 other 对象
__le__(slef,other)	判断 self 对象是否小于或者等于 other 对象
__eq__(slef,other)	判断 self 对象是否等于 other 对象
__call__(self, * args)	把实例对象作为函数调用

6.3.2 对象

类是具有相同属性和行为的一组对象的集合,它只是一个抽象的概念,而对象指的是某个具体的事物,在本节开始时已经说过,现实世界中客观存在的一个点、一个人、一辆车等都可以视为一个个对象,它们具有自己独有的属性:一个点有自己的坐标,人有自己的身份证号,而车有车牌号。它们又有一些可以被改变的属性,例如当车被卖之后它的主人就发生了更换,主人这个属性就是可以变的;同时,对象也会有自己的行为,继续以上面的汽车为例子,汽车可以有启动、前进、后退等行为。不同的属性与行为的组合构成了一个个对象。在使用对象时需要创建一个或者多个这个类的对象实例,这个过程称

为实例化。下面通过例子来更好地理解类和对象的关系。

类是抽象的概念，类实现动态方法和静态属性(即实例变量、类变量)的定义。但是，类在内存是不存储的，如果有类变量，内存仅存储该类变量。对象是真实存在的实体，它将复制类中定义的所有方法(包括内置方法和普通方法)和实例变量，但不复制类变量。每个对象都有自己的一套内存空间，存储各自的实际实例变量值。不同对象间不共享任何方法或实例变量。每个对象在程序中从创建到销毁都分别经历了“出生”“存活”和“消亡”三个阶段。

(1) 出生：即类的实例化，对象名＝ 类名([arg0，…])。在实例化过程中，先用__new__()方法创建原始对象，该方法自动调用且在类中不用写明，再用__init__()方法初始化该对象，该方法自动调用但在类中要写明。

(2) 存活：在运行时执行类中各种普通的方法。

(3) 消亡：Python 采用垃圾回收机制来清理不再使用的对象。__del__()方法释放对象，该方法可以在对象被销毁时自动调用，也可以通过对象显示调用，但在类中写明。

【例 6-11】 从下面程序示例中体会实例化一个对象的过程。

```
import math
class Circle():
    count = 0
    def __init__(self,x,y,r):
        self.x = x
        self.y = y
        self.r = r
        Circle.count += 1
        print('current num is',Circle.count)
    def distance(self):
    x0 = 0
    y0 = 0
    d = math.sqrt((self.x - x0) * * 2 + (self.y - y0) * * 2)
    return d
```

```
    def __del__(self):
        Circle.count -=1
        print('current num is',Circle.count)
>>>c1 =Circle(10,20,5)
>>>print('dis is',c1.distance())
>>>c2 =Circle(1,2,4)
>>>print('dis is',c2.distance())
>>>c1.__del__()
>>>c2.__del__()
```

在本例中,c1 = Circle(10,20,5)表示创建了一个 Circle 对象,print 语句则是函数在生命周期内的行为,最后执行了__del__()表示对象被销毁,清晰地展现了一个对象的生命周期。

6.3.3 继承

在现实世界中,类和类之间不是孤立存在的,它们之间存在一般和特殊的关系,通过分析这种关系,可以将所有类组织成为一个层次结构,称为类层次。例如,人分为学生和教师,学生又可以分为研究生和本科生,等等。为了描述这种一般与特殊的类间关系,面向对象程序设计提供相应的类定义方式,可以从已有类派生出新类,新类拥有已有类的所有属性和方法,这个过程称为继承,被继承的类称为父类或超类,继承的类称为子类。子类由父类生成,因而它拥有父类的一切属性,即静态属性和动态属性。但是,子类又具有自己的特点,可以改写父类属性,也可以创建新的属性。

在面向对象程序设计时,通常先定义父类,然后定义子类,并让子类继承父类;同时,在子类中再重新定义父类属性或添加自己的属性。继承实现了代码重用,子类可以不必重复定义从父类中继承来的属性,从而简化了程序。子类继承父类格式为:

```
class 子类名(父类名):
    方法定义 1:
    …
    方法定义 2:
…
```

子类通过以下语法继承父类方法：

子类内部:父类名.方法([arg])
子类外部:对象名.方法([arg])

【例 6-12】 下面的示例说明了类的继承关系。学生和老师都继承人类,但各自有特有的属性和方法：老师有工资,可以讲课;学生有班级,可以听课。

```
class Person():
    def __init__(self,name,pid):
            self.name = name
        self.pid = pid

    def info(self):
        print(self.name,'id is', self.pid)

class Teacher(Person):                        #继承 Person
    def __init__(self,name,pid,salary):
        Person.__init__(self,name,pid)
        self.salary = salary
        print('inherit class Person()')

    def teachClass(self,className):
        print(self.name,'teaches',className)

    class Student(Person):                    #继承 Person
        def __init__(self,name,pid,classNum):
        person.__init__(self,name,pid)
        self.classnum = classNum
        print('inherit class Person()')

    def takeLessons(self,lessonName):
        print(self.name,'takes',lessonName)
```

```
>>>t =Teacher('Shirley',123,4500)
>>>print(t.info())
>>>t.teachClass('computer')

>>>s =Student('Catherine ',2342,3806)
>>>print(s.info())
>>>s.takeLessons('computer')
```

由上面的例子可以看出,在定义子类时所使用的语法是 class 子类名(父类名)。当我们要在子类内部继承父类的方法是父类名.方法([arg]),继承父类公有实例变量的方法是 self.变量名,在子类外部继承父类的方法是对象名.方法([arg]),继承父类公有实例变量的方法是对象名.变量名,子类继承父类变量的方法是父类名.变量名。

当 Teacher 或 Student 类继承 Person 类时,子类能继承父类所有的属性和方法;同时,子类可以重新定义父类的某些属性和方法或者创建属于自己的方法。例如,Teacher 和 Student 都重写了父类的__init__方法,这种行为称为覆盖,Teacher 类的 teachClass 方法在 Person 类里就没有定义,而是在 Teacher 类里自己定义的。

子类仍可以定义自己的子类,即父亲的孙类。下面继续上一个的例子,教师有来自美国说英语的英文教师和来自中国说汉语的中文教师,可以表示为图 6-5 所示的结构。

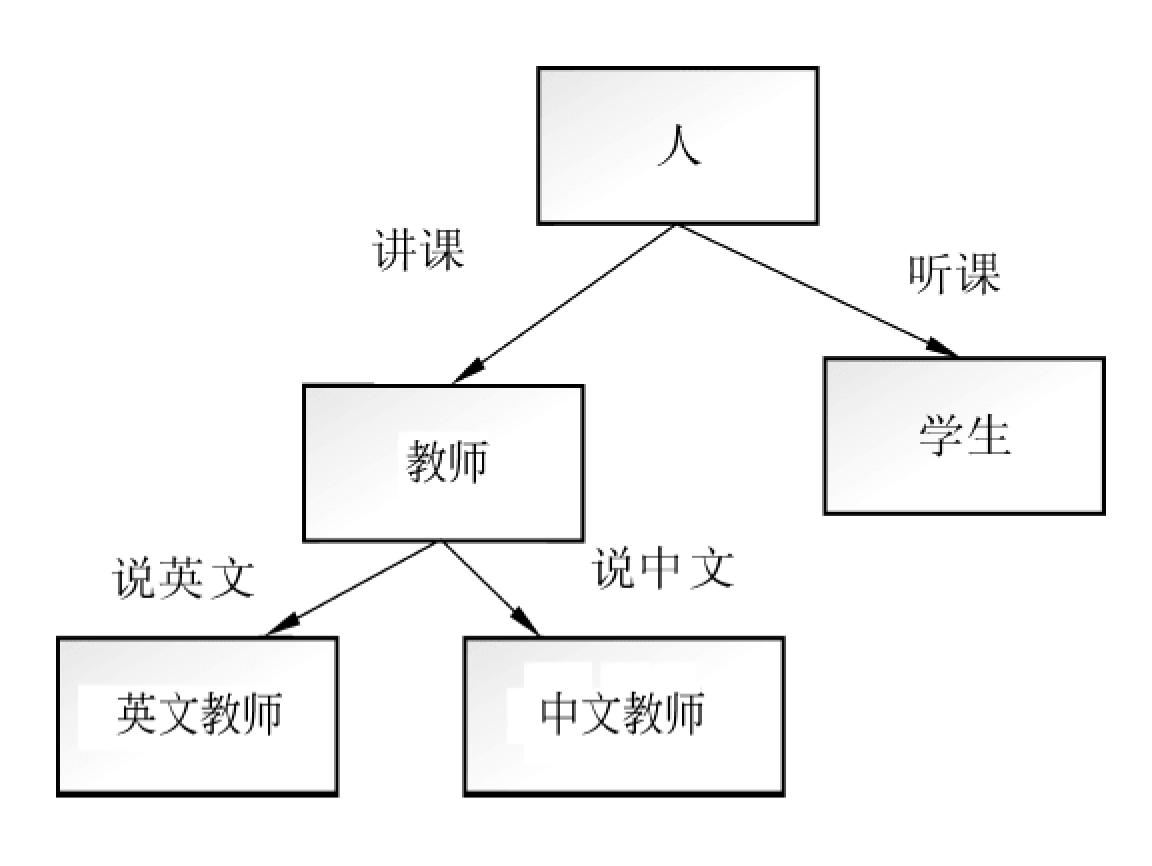


图 6-5　类的继承实例

【例 6-13】 例 6-12 中 Person 子类 Teacher 创建自己的子类。

子类代码如下：

```
class Person():
        ...
class Teacher(Person):                 #继承 Person
        ...
class ChineseTeacher(Teacher):         #继承 Teacher,即 Person 的孙类
    def __init__(self,name,pid,salary,language):
        teacher.__init__(self,name,pid,salary)
        self.language = language
        print('inherit class teacher()')
    def speaking(self):
        print(self.name,'speaks',self.language)

class AmericanTeacher(Teacher):        #继承 Teacher,即 Person 的孙类
    def __init__(self,name,pid,salary,language):
        teacher.__init__(self,name,pid,salary)
        self.language = language
        print('inherit class teacher()')
    def speaking(self):
        print(self.name,'speaks',self.language)
```

子类 ChineseTeacher、AmericanTeacher 即为 Person 的孙类。

继承还有一种情况，即一个子类可以同时继承多个父类，称这种继承为多重继承。

例如，在一个家庭中，孩子继承了爸爸的双眼皮，同时继承了妈妈的宽额头，若把他们当成类就会出现子类继承两个父类的情况。

【例 6-14】 下面示例中 mySelf 同时继承了 myFather 和 myMother 两类。

```
class MyFather:
    def __init__(self):
        self.eyes = '爸爸的眼睛是双眼皮'
```

```
            print(self.eyes)

class MyMother:
    def __init__(self):
            self.forehead ='妈妈的额头有点宽'

class MySelf(MyFather,MyMother):   #同时继承了父亲类和母亲类
    def __init__(self,face):
            print('我继承了')
            MyFather.__init__(self)
            print('我继承了')
            MyMother.__init__(self)
            self.face =face
            print('可是我的脸型是%s'%(self.face))

mylook =mySelf('圆脸')
```

mySelf 类继承了 myFather 类和 myMother 类的__init__方法，是它们的子类，可以看出子类多重继承的格式是 class 类名(父类 1，父类 2，…)。运行时发现，输出结果中子类并没有继承 myMother 类的__init__方法，这是因为当使用多重继承时，如果一个方法从多个父类继承，那么父类的顺序不能颠倒，先继承类的方法会重写后继承类的方法。

虽然多重继承是非常有用的工具，但是除非特别熟悉它，否则应尽量少使用，以免出现不必要的错误。

6.3.4 多态

多态(polymorphic)来自希腊语，意思是“有很多形式”。多态意味着我们可以在不知道变量类型的情况下仍能对其进行操作，它也会根据对象(或类)类型的不同而表现出不同的行为。多态是面向对象程序设计(OOP)的一个重要特征。

首先来看下面一个简单的例子：

```
>>>a=1
>>>b=2
>>>print(a+b)
>>>a="Hello"
>>>b="World"
>>>print(a+b)
```

在上面例子中，我们不知道加号“＋”运算符左右两个变量是什么类型，当给的是 int 类型时，它就进行加法运算。当给的是字符串类型时，它就返回两个字符串拼接的结果。也就是说，根据变量类型的不同表现出不同的形态，即为多态，在运行时确定其状态，在编译阶段无法确定其类型，根据输入的不同，采用不同的方法。

在 Python 中，很多内建函数和方法都有多态的性质。例如，上例中的“＋”就是运算符多态的体现。我们之前用过 Python 的内置函数 len，它就是一个多态函数，用来计算一个序列所有元素的个数，所有其元素是序列的对象（如字符串、元组、列表）都可以使用。其使用示例如下：

```
#计算字符串的长度：
s ="Hello world!"
>>>len(s)
```

结果为：

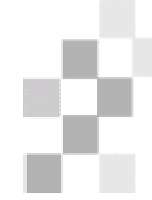
```
12
```

```
#计算列表的元素个数
l =['h','e','l','l','o']
>>>len(l)
```

结果为：

```
5
```

可以看出，在第一个例子中 len 用于求字符串的长度，而第二个例子中则用于求列表



元素的个数。在使用前,我们并不知道参数的类型,在运行时确定参数类型,这很好地体现出 len 函数的多态的性质。如果函数内部所有的操作都支持某种类型,那么这个函数就可以用于那种类型,从这个角度说,多态可以促进代码的复用。

6.4 模块化编程思想

在生活中,在现代社会中,模块化的思想在人们的日常生活中处处都有体现,搭建房屋、建造船舶、组装汽车以及设计电子器件时也常常是模块化的设计。模块是指能够提供特定功能的相对独立的单元。通过模块化思想可以不必重复去做相同的事情。在前面的介绍中我们也知道,在编程的过程中代码的重用是非常重要的,在编写程序时很多时候人们都用到相同功能的代码,如果不学会利用之前的程序就会浪费很多时间。在 Python 中提供了多种代码重用的方式,我们已经学过面向过程编程和面向对象编程,它们都是模块化编程思想的体现,下面来详细介绍模块。

6.4.1 模块

在之前的例子中已经用到了一些模块。例如 math,这中间包括了数学计算中常用的一些函数。当编写的程序规模变大,想把所有程序都存储在一个文件当中时,这并不适用,现在典型的方法是将程序不同部分存储在不同的文件当中,Python module 提供了这样的编程方式,可以在多个文件中构建程序代码,从而完成比较复杂程序的协同。

模块化编程将软件分解为若干个独立的、可替换的、具有预定功能的模块,每个模块实现一个功能,各模块通过接口(输入输出)组合在一起,形成最终的程序。这样,可以使程序易设计、易实现、易测试、易维护和可重用。

在 Python 中的模块是一个以.py 结尾的 Python 代码文件,是把一组相关函数、类或代码保存在一个文件中形成的。其中,类和函数可以有 0 到多个。Python 中常用的模块有 math、random、string、os、sys 以及与网络处理相关的 httplib、ftplib 和 maillib 等。在 Windows 系统环境下,Python 公共模块一般存放在 C:\python34\lib。

定义一个模块只需要把文件保存为 *.py 文件即可，但要注意文件名不能以中文命名，否则导入时会报错。当需要使用该模块时，只需要调用模块，用"import＋文件名.函数名、文件名.类名"命令即可。

这里要注意 import 导入文件路径的问题，Python 导入文件的默认路径是 sys.path，所以，导入的模块要么放置在与输入它的程序的同一个目录中，要么用 sys.path.append 命令将所要导入文件的绝对路径添加到默认路径中，如 sys.path.append('c：/test')。

任何包含 Python 代码的文件都可以作为模块导入。

【例 6-15】 模块的导入实例。

现在有一个文件 wc.py，其代码如下：

```
def linecount(filename):
    count=0
    for line in open(filename):
        count+=1
return count
```

可以通过下面的语句导入并使用这个模块，该模块提供了一个函数 linecount( )，这样就能通过下面的方式调用这个函数，统计任意一个文件的行数。

```
>>>import wc
>>>lines=wc.linecount('wc.py')
>>>print(lines)
>>>5
```

Python 有很多现成可以用的模块，可是这么多模块使用和维护起来还是不太方便。为了组织好模块，可以将这些模块分组为包(package)。包就是由很多 *.py 组成的目录，即文件夹。Python 中有很多包，它们的默认路径为 Python34\Lib。例如，测试常用的 unittest 包、日志分析常用的 logging 包等，同时还有现在机器学习常用的 numpy、networkx 等。

在编写大型程序时，常常需要将模块封装为包。例如，现在我们自己创建一个名为

utils 的包,首先要新建一个空的文件夹并命名为 utils,在文件夹中添加__init__. py 文件,然后添加其他模块或子文件夹,并将其放在 Python34\Lib 目录下,其效果如图 6-6 所示。

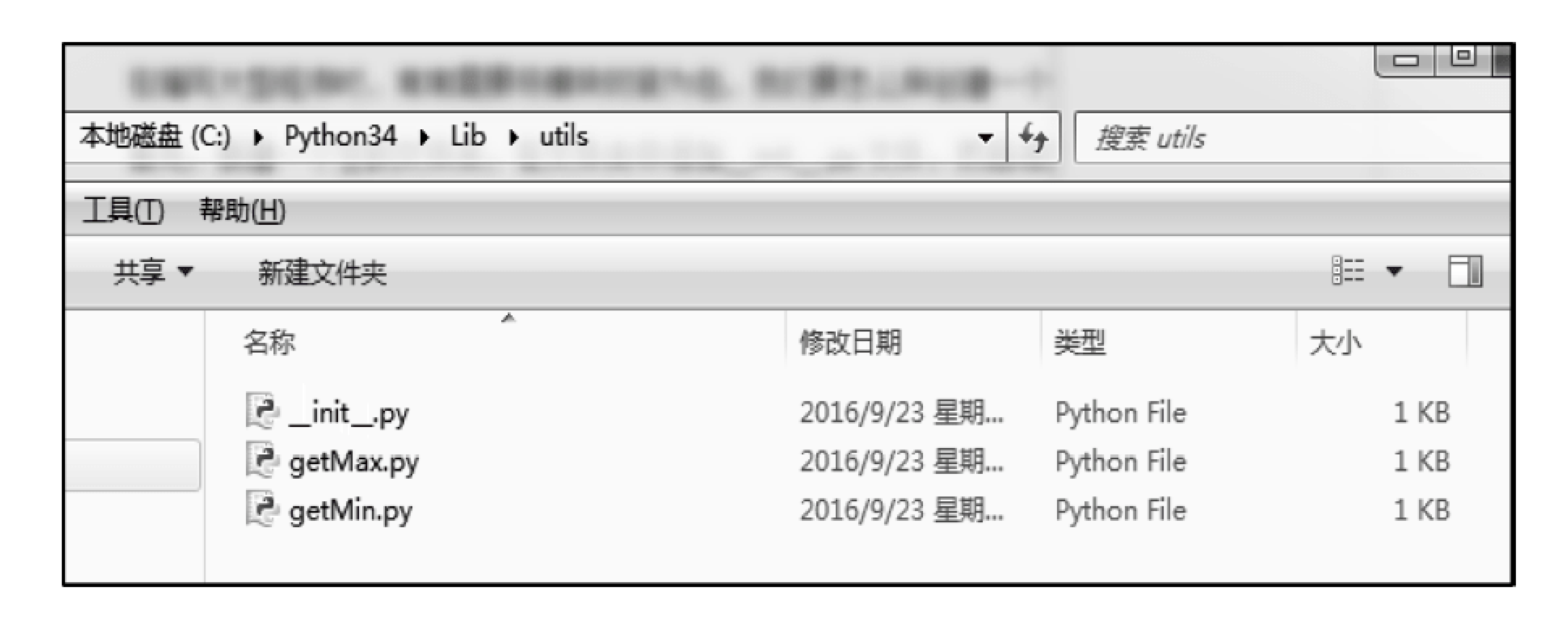


图 6-6 包的建立

_ _init_ _. py 控制着包的导入行为,如果其为空,则仅导入包,什么都不做,文件夹下只有存在_ _init_ _. py 文件,这个文件夹才能被 python 视为包,我们才能导入这个包下的模块(module)。想要在包中添加内容其格式为添加内容:_ _all_ _ = [文件夹中文件或子文件夹名称]。图 6-6 中的 getMax 和 getMin 是用来保存求最大值和最小值的函数的文件。

包建好了,怎样才可以使用已经创建好的包呢?有两种方法:第一种方法是把包放置到 Python34\Lib 中,使用时直接引用;第二种方法是将包所在的路径添加到 sys. path 中,再引用。

在前面我们已经把包放到默认路径下了,可以用如下方法使用它:

```
from utils import getMax
from utils import getMin
>>>print (getMax.max(2, 3))
>>>print (getMin.min(3, 2))
```

结果分别为:

```
3, 2
```

【例 6-16】 从标准库导入模块。

```
import math
def sq():
    number =int(input('Enter a number: '))
    square_root =math.sqrt(number)
    print ('The square root of',number,'is',square_root)
>>>sq()
```

可以看到如下结果：

```
Enter a number: 645
The square root of 645 is 25.3968501984
```

这里引入了 Python 里的 math 模块，通过调用 math. sqrt 求出所输入数字的平方根。

Python 提供了下面一些常见的模块。

1. math 模块

math 模块是 Python 中的数字处理模块，它实现了许多对浮点数的数学运算函数，常见的有三角函数 sin(x)、cos(x)、tan(x)、asin(x)等，还有常用的向上取整 ceil()、向下取整 floor()等函数。这些函数极大地方便了人们编程时对数字的处理。

2. random 模块

Python 中 random 模块包含返回随机数的函数，可以用于模拟或用于任何随机产生和随机输出的程序。表 6-3 给出了 random 模块中的一些重要函数。

表 6-3　random 模块中的一些重要函数

函　数	描　述
randomm()	用于生成一个[0,1)的实数
uniform(a, b)	用于生成一个[a,b)内的随机实数
randint(a, b)	用于生成一个[a,b)内的整数
randrange([start], stop,[step])	从指定范围内，按指定基数(step)递增的集合中获取一个随机数
choice(seq)	从有序类型(seq)等返回任意元素

【例 6-17】 在聚会时,大家喜欢玩掷骰子游戏,现在编写一个程序,要求用户选择骰子的个数,然后投掷骰子,最后输出点数之和。

```
from random import randrange                        #导入模块
>>>num = input("How many dice?")                    #输入骰子个数
>>>sum = 0
>>>for I in range(num): sum+ = randrange(6)+1
                                                    #随机生成[0,5]的随机数
>>>print('The result is',sum)
```

这里需要输入一个随机的骰子个数,当随意输入一个骰子个数后,每个骰子随机生成一个点数,并输出其点数之和。

3. os 模块

os 模块包含普遍的操作系统功能,如果希望自己的程序能够与平台无关的话,这个模块尤为重要。os 模块为我们提供了访问多个操作系统服务的功能。os 模块包括的内容很多,表 6-4 列出了 os 模块中一些重要的函数和变量。

表 6-4 os 模块中一些重要的函数和变量

函数/变量	描 述
name	显示正在使用的平台
linesep	给出当前平台使用的行终止符
getenv()和 putenv()	分别用来读取和设置环境变量
remove()	用来删除一个文件
system()	用来运行 shell 命令
getcwd()	得到当前工作目录,即当前 Python 脚本工作的目录路径

4. sys 模块

sys 模块包含与系统对应的功能,能够访问与 Python 解释器联系紧密的变量和函数。表 6-5 列出了 sys 模块中一些重要的函数和变量。

表 6-5　sys 模块中一些重要的函数和变量

函数/变量	描　　述
version	显示 Python 版本号
path	查找模块所在目录的目录名列表
argv	命令行参数，包括脚本名称
exit([arg])	退出当前程序，可选参数为给定的返回值或错误信息

下面介绍命令行参数，在通过命令行执行 Python 脚本时可能会在后面加上命令行参数。这些参数放置在 sys.argv 参数列表中，可以通过如例 6-16 所示代码形式将其反打印出来。

【例 6-18】 反打印 argv 参数。

```
#reverseargs.py
import sys
args=sys.argv[1:v]
args.reverse()
print (' '.join(sys.argv[1:]))
```

```
$python  reverseargs.py  this is a test
```

其输出结果为：

```
test a is this
```

5. string 模块

任何语言都离不开字符，都会涉及对字符的操作，尤其脚本语言更是频繁使用字符，不管是生产环境还是考试，都要面对字符串的操作，string 模块提供了一些用于处理字符串类型的函数。表 6-6 列出了 string 模块中一些重要的函数和变量。

6. time 模块

time 模块是与 Python 中事件处理相关的模块，主要实现以下功能：获得当前时间、

操作时间和日期、从字符串读取时间以及格式化时间为字符串。

表 6-6　string 模块中一些重要的函数和变量

函　　数	说　　明
capitalize(string)	把字符串的首个字符替换成大字
lower(string)	把字符串转化为小写
upper(string)	把字符串转化为大写
replace(string,old,new[,maxsplit])	把字符串中的 old 替换成 new,maxsplit 用于设置可替换的个数,默认替换所有 old
split(string,sep=None,maxsplit=-1)	从 string 字符串中返回一个列表,以 sep 的值为分界符
join(string[,sep])	返回用 sep 连接的字串,默认的 sep 是空格

1）时间戳(timestamp)

通常时间戳表示的是从 1970 年 1 月 1 日 00:00:00 开始按秒计算的偏移量。我们运行"type(time. time())",返回的是 float 类型。

2）元组(struct_time)

struct_time 元组共有 9 个元素：年,月,日,时,分,秒,一年中的第几周,一年中的第几天,是否是夏令时。表 6-7 列出了 struct_time 元组元素的索引含义。

表 6-7　struct_time 元组元素的索引含义

索引(Index)	属性(Attribute)	值(Values)
0	tm_year(年)	如 2016
1	tm_mon(月)	1～12
2	tm_mday(日)	1～31
3	tm_hour(时)	0～23
4	tm_min(分)	0～59
5	tm_sec(秒)	0～61
6	tm_wday(一年中的第几周)	0～6(0 表示周日)
7	tm_yday(一年中的第几天)	1～366
8	tm_isdst(是否是夏令时)	默认为-1

可以看到，秒的范围为 0～61，这是是为了应付闰秒和双闰秒。夏令时的数字是布尔值(true 或 false)time 模块就会工作正常。time 模块中重要的函数如表 6-8 所示。

表 6-8　time 模块中重要的函数

函　　数	描　　述
asctime([tuple])	将一个时间元组转换成字符串
localtime([seconds])	将秒数转化为日期元组，以本地时间为准
mktime(tuple)	将时间元组转化为本地时间
sleep(seconds)	休眠 seconds 秒
strptime(string, format)	将时间字符串解析为时间元组
time()	当前时间(新纪元开始后的秒数，以 UTC 为准)

注：新纪元是一个与平台相关的年份，例如 UNIX 的新纪年是 1970 年。

本节只列举了一些常用的函数和其用法，想要更好地了解 Python 中的模块，可以查阅 Python 在线文档。

6.4.2　文件

到现在为止，程序与外部的交互只是通过 input 和 print 函数实现的，其他几乎与外界没有交互。然而，在实际编程过程中，我们常常遇到需要使用其他格式输入的情况，比较常用的一种格式就是文件，文件是指存储在外存储器上的某类信息结合体，可以是 .txt 格式的文本文档，也可以是 .mp3 格式的音乐文件，还可以是 .mp4 格式的视频文件，等等。人们用各种不同格式的文件来存储各种不同的数据，通常不同内容的数据也有不同的格式。一般来说，我们的程序源码是可以在编辑器中看到，能看到的文件被认为是一个字符序列，这种存储格式称为文本格式。读写文本文件是程序维护数据最简单的方法之一。

实际上，文本文件存储的是每个字符的 ASCII 码。如果想在文本里存储一个数字 100，存储的就是 1、1、0 三个字符的 ASCII 码，即 0x0000064。另一种更常用的是二进制格式存储，它把 100 转化为对应的二进制编码，即 0x00000064 存储，这时候是不能用文

本编辑器看到的，此时显示的就是乱码。两种显示方法显然存在较大差别。

在 Python 中要访问磁盘文件，必须打开 Python Shell 与磁盘上存放的文件直接的链接。这个链接是通过一个文件对象来完成的，在建立连接时就生成该文件对象，通过对对象的操作可以在程序运行期间存储数据，处理来自其他程序的数据。Python 中对文件的操作分为打开文件、对文件的读写操作和关闭文件。

1. 打开文件

Python 中使用 open 函数命令打开一个文件，建立连接，并且返回代表连接的文件对象，通过文件对象执行文件上所有的后续操作，文件对象也称为文件操作符或文件流。open 函数语法如下：

```
open(name[,mode[,buffering]])
```

其中，只有文件名不能缺少，模式(mode)和缓冲(buffering)是可选的。如果 open 函数只带一个文件名参数，那我们可以获得能获取文件内容的文件对象，这时候的默认效果是只读模式。如果想要写入内容，则必须提供一个模式参数 w 来显示声明。+参数可以用到任何模式中，指明读和写都是允许的。r+即表示打开的文本文件同时支持读写操作。表 6-9 列出了 open 函数中模式参数及其意义。

表 6-9　open 函数中模式参数及其意义

参数值	描述
r	读模式
w	写模式
a	追加模式
b	二进制模式(可添加到其他模式)
+	读写模式(可添加到其他模式)

open 函数中还存在第三个参数控制着文件中的缓冲，参数为 0 或 False，I/O 就是无缓存的，如果是 1 或 True 则是有缓存的，大于 1 的数字代表缓存大小。当参数为－1 时，则代表使用默认缓存大小。

现在有一个名为 data 的二进制文件，首先要将其打开，其语句如下：

```
>>>f =open('data', 'rb+')    #以二进制读取方式读取 data 文件
```

2. 读取文件

当打开一个文件后，就需要对文件进行操作，读文件是我们经常用到的一个操作。读取文件时有三种方式，分别为 read()、readline()、readlines()，其文件读取操作方式及其意义如表 6-10 所示。

表 6-10　文件读取操作方式及其意义

操作方式	描　述
read()	每次读取整个文件，它通常用于将文件内容放到一个字符串变量中
readline()	每次读取一行文件，返回一行字符串
readlines()	每次读取整个文件，返回包含文件所有内容的字符串列表，每个元素是一行的字符串

3. 写入文件

对文件进行写操作也是经常用到的，对文件写操作有两种方式：write() 和 write (inel)，其文件写入操作方式及其意义如表 6-11 所示。

表 6-11　文件写入操作方式及其意义

操 作 方 式	描　述
write()	写入内容后光标在行末不会换行，下次写会接着这行写
writeline()	写入内容后光标跳到下一行起始位置，下次写会在新行

假如现在要对刚才打开的 data 文件进行写操作并且读取它的前 4 个字符，则操作如下：

```
>>>f.write('101001')        #写入字符'101001'
>>>f.readline(4)            #读取前 4 个字符
>>>f.close()                #关闭文件
```

4. 关闭文件

当完成对一个文件的读写操作后并没有结束，在程序的最后还要用 close()方法关闭文件，尽管 Python 中文件对象关闭后会自动关闭文件，但记得关闭文件是没有坏处的，还可以避免浪费系统资源，有助于养成良好的编程习惯。

将上面例子连起来就是对一个文件的基本读写操作。

【例 6-19】 读写二进制文件。

现在有一个二进制文件 data，编程将其读入，并写入二进制字符串，最后得到它的前 4 个元素。

```
>>>f =open('data', 'rb+')       #以二进制读取方式读取 data 文件
>>>f.write('101001')            #写入字符'101001'
>>>f.readline(4)                #读取前 4 个字符
>>>f.close()                    #关闭文件
```

以上例子给出了文件读写的基本流程，但是当尝试读取和写入文件时，很多环节都有可能出错。例如，当尝试打开一个并不存在的文件，或者尝试去写入一个只读型的文本，或者尝试打开一个目录用于文件读取，这时候就会得到一个 IOError，出现文件读取错误，程序没有办法按照例 6-19 正确的流程执行。要避免包括以上提到的各种文件读取的可能错误，使程序不至于执行不下去，最好用 try-catch 去捕获可能发生的错误。

【例 6-20】 读写文本文件。

```
try:
    f=open('d:/hello_python.txt','w')
    f.write('hello my friend python!')
except IOError:
    print('IOError')
finally:
    f.close()


try:
```

```
    f=open('d:\hello_ python.txt','r')
    print(f.read())
    f.close()                    #获取在当前文件中目前所处的位置,起始值为 0
except ValueError as ioerror:
    print('File alread closed {0}'.format(type(ioerror)))
finally:
    print('operation end')
```

习　　题

1. 任意给出一个整数,试着编写一个函数,判断它是否为素数,其返回值为 true 或者 false。

2. 通过键盘输入 n(n＞2)以及 n 元一次方程组的各项参数,采用递归算法,利用加减消元法求出各项未知数,并且参照示例给出结果。

例如,输入方程

$$\begin{cases} x1+2x2+3x3=4 & ① \\ x1+3x2+5x3=7 & ② \\ 2x1+3x2+5x3=6 & ③ \end{cases}$$

输出结果：x1＝－1,x2＝1,x3＝1。

提示：加减消元法是利用等式的性质,使方程组中两个方程中的某一个未知数前的系数的绝对值相等,然后把两个方程相加或相减,以消去这个未知数,使方程只含有一个未知数而得以求解。

以上面的方程为例,

②－①得 x2＋2x3＝3

②×2－③得 3x2＋5x3＝8

再次进行加减消元得 x3＝1,代入上式得 x2＝1,代入原式得 x1＝－1。

3. 实现给定区间的二分查找。具体要求如下：

(1) 接收用户从键盘输入一个 N 个数的有序整数序列。

(2) 用户输入一个数，在该有序整数序列中用二分搜索查找该数。若找到，则输出其在整数序列中的位置编号；若未找到，则输出“NOT FOUND!”。

4. 某国为了防御敌国的导弹袭击，研发出一种导弹拦截系统。但是，这种拦截系统有一个缺陷，虽然它的第一发炮弹能够到达任意的高度，但以后每一发炮弹都不能高于前一发炮弹的高度。某天，雷达捕捉到敌国的导弹来袭，由于该系统还在试用阶段，所以该套系统有可能不能拦截所有的导弹。

输入导弹依次飞来的高度，雷达给出的高度不大于 30000 的正整数。计算要拦截所有的导弹，最少需要配备多少套这种导弹拦截系统。

输入：导弹数 n 和 n 颗导弹依次飞来的高度，$1 \leqslant n \leqslant 1000$。

输出：要拦截所有的导弹最少配备的系统数。

5. 设计一个名为 Rectangle 的类来表示矩形，这个类包括宽 width 和高 height 两个类变量，长和宽初始值分别为 3 和 4，并且允许分别改变其值，并能分别得到它的周长和面积。

6. 当前目录下有一个文件名为 score.txt 的文本文件，存放着某班学生的学号、姓名、英语课成绩（第 3 列）和语文课成绩（第 4 列）。请编程完成下列功能：

(1) 分别求出这个班英语和语文的平均分（保留 1 位小数）并输出。

(2) 找出两门课都不及格（<60 分）的学生，输出其学号和各科成绩。

(3) 找出两门课的平均分在 85 分以上的学生，输出其学号和各科成绩。

试着用三个函数分别实现以上要求。